

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

**APPLICATION PROGRAMMER'S INTERFACE (API) FOR
HETEROGENEOUS LANGUAGE ENVIRONMENT AND
UPGRADING THE LEGACY EMBEDDED SOFTWARE**

by

Theng C. Moua

September 2001

Thesis Advisor:
Second Reader:

Valdis Berzins
Jun Ge

Approved for public release; distribution is unlimited.

Report Documentation Page		
Report Date 30 Sep 2001	Report Type N/A	Dates Covered (from... to) -
Title and Subtitle Application Programmer's Interface (API) for Heterogeneous Language Environment and upgrading the Legacy Embedded Software	Contract Number	
	Grant Number	
	Program Element Number	
Author(s) Moua. Theng C.	Project Number	
	Task Number	
	Work Unit Number	
Performing Organization Name(s) and Address(es) Research Office Naval Postgraduate School Monterey, Ca 93943-5138	Performing Organization Report Number	
Sponsoring/Monitoring Agency Name(s) and Address(es)	Sponsor/Monitor's Acronym(s)	
	Sponsor/Monitor's Report Number(s)	
Distribution/Availability Statement Approved for public release, distribution unlimited		
Supplementary Notes		
Abstract		
Subject Terms		
Report Classification unclassified	Classification of this page unclassified	
Classification of Abstract unclassified	Limitation of Abstract UU	
Number of Pages 116		

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE September 2001	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE: Title (Mix case letters) Application Programmer's Interface (API) for Heterogeneous Language Environment and upgrading the Legacy Embedded Software			5. FUNDING NUMBERS	
6. AUTHOR(S) Moua, Theng C.				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) <p>Legacy software systems in the Department of Defense (DoD) have been evolving and are becoming increasingly complex while providing more functionality. The shortage of original software designs, lack of corporate knowledge and software design documentation, unsupported programming languages, and obsolete real-time operating system and development tools have become critical issues for the acquisition community. Consequently, these systems are now very costly to maintain and upgrade in order to meet current and future functional and nonfunctional requirements.</p> <p>This thesis proposes a new interoperability model for re-engineering of old procedural software of the Multifunctional Information Distributed System Low Volume Terminal (MIDS-LVT) to a modern object-oriented architecture. In the MIDS-LVT modernization acquisition strategy, only one Computer Software Configuration Item (CSCI) at a time will be redesigned into an object-oriented program while interoperability with other unmodified CSCIs in the MIDS-LVT distributed environment must be maintained. Using this model, each legacy CSCI component can be redesigned independently without affecting the others.</p>				
14. SUBJECT TERMS Distributed Systems, Embedded Systems, Design Pattern, Architecture, Application Programmer' Interface, Interoperability, Protocol, Temporal, Software			15. NUMBER OF PAGES	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited.

**APPLICATION PROGRAMMER'S INTERFACE (API) FOR HETEROGENEOUS
LANGUAGE ENVIRONMENT AND UPGRADING THE LEGACY EMBEDDED
SOFTWARE**

Theng C. Moua
B.S.E.E., San Diego State University, 1985

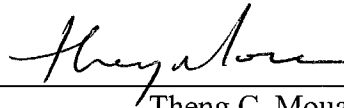
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

from the

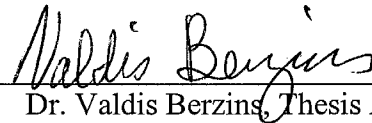
**NAVAL POSTGRADUATE SCHOOL
September 2001**

Author:



Theng C. Moua

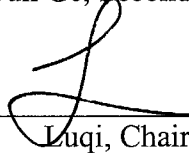
Approved by:



Dr. Valdis Berzins, Thesis Advisor



Dr. Jun Ge, Second Reader



Luqi, Chair
Software Engineering Curriculum

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Legacy software systems in the Department of Defense (DoD) have been evolving and are becoming increasingly complex and are becoming increasingly complex while providing more functionality. The shortage of original software designs, lack of corporate knowledge and software design documentation, unsupported programming languages, and obsolete real-time operating system and development tools have become critical issues for the acquisition community. Consequently, these systems are now very costly to maintain and upgrade in order to meet current and future functional and nonfunctional requirements.

This thesis proposes a new interoperability model for re-engineering of old procedural software of the Multifunctional Information Distributed System Low Volume Terminal (MIDS-LVT) to a modern object-oriented architecture. In the MIDS-LVT modernization acquisition strategy, only one Computer Software Configuration Item (CSCI) component at a time will be redesigned into an object-oriented program while interoperability with other unmodified CSCIs in the MIDS-LVT distributed environment must be maintained. Using this model, each legacy CSCI component can be redesigned independently without affecting the others.

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
II.	BACKGROUND.....	7
A.	MIDS-LVT SYSTEM.....	7
1.	<i>System Architecture</i>	8
2.	<i>Re-engineering Effort</i>	12
B.	DISTRIBUTED REAL-TIME SYSTEMS.....	13
1.	<i>Real-Time System</i>	13
a.	Embedded System.....	13
b.	Timing Constraint.....	14
c.	Real-Time Control.....	14
d.	Time-Driven System.....	14
e.	Reactive System.....	14
2.	<i>Distributed System</i>	14
a.	Asynchronous Model.....	15
b.	Synchronous Model.....	15
3.	<i>Real-Time Operating System</i>	15
C.	BENEFITS OF REAL-TIME DISTRIBUTED APPLICATIONS.....	16
1.	<i>Load Balancing</i>	16
2.	<i>Improved Response Time</i>	16
3.	<i>More Economical</i>	16
4.	<i>Improve Scalability</i>	17
5.	<i>Extendable</i>	17
D.	CHALLENGES OF DEVELOPING REAL-TIME DISTRIBUTED APPLICATIONS.....	17
1.	<i>Concurrency and Schedulability</i>	17
2.	<i>Timing Constraint</i>	18
3.	<i>Dynamic Behavior</i>	19
4.	<i>Correctness and Robustness</i>	19
E.	INTER-PROCESSOR COMMUNICATION.....	19
1.	<i>Shared Memory</i>	20
2.	<i>Pipe</i>	21
3.	<i>Distributed Shared Memory</i>	21
4.	<i>Signals</i>	21
5.	<i>Alarms</i>	22
6.	<i>Mutual Exclusion</i>	22
F.	OBJECT-ORIENTED METHODOLOGY.....	23
1.	<i>Abstraction</i>	23
2.	<i>Encapsulation/Information Hiding</i>	24
3.	<i>Inheritance</i>	24
4.	<i>Polymorphism/Dynamic Binding</i>	24
G.	DESIGN PATTERNS.....	24

1.	<i>Facade Pattern</i>	25
2.	<i>Decorator Pattern</i>	25
3.	<i>Strategy Pattern</i>	25
4.	<i>Proxy Pattern</i>	26
H.	GENERATIVE PROGRAMMING	26
I.	CORBA TECHNOLOGY	27
1.	<i>Properties</i>	27
a.	Heterogeneous Environment.....	27
b.	Language Independent	27
c.	Location Independent.....	27
2.	<i>Object Request Broker</i>	28
3.	<i>Interface Definition Language</i>	28
4.	<i>Real-Time/Minimum CORBA</i>	29
a.	Minimum CORBA	29
b.	Real-Time CORBA	29
J.	INTEROPERABILITY TECHNIQUES	31
F.	SUMMARY	32
III.	SPECIFICATION AND MODEL	35
A.	MIDS-LVT SOFTWARE ARCHITECTURE	35
1.	<i>CSCI Architecture View</i>	37
2.	<i>Hardware Wrapper</i>	39
B.	INTEROPERABILITY MODEL	43
1.	<i>Interface Specification</i>	43
2.	<i>Protocol Specification</i>	44
3.	<i>Temporal Specification</i>	45
C.	LOW LEVEL PROTOCOL SPECIFICATION	46
1.	<i>Handshake Word</i>	47
2.	<i>Pointers</i>	48
3.	<i>Data Transfer Blocks</i>	48
D.	TEMPORAL SPECIFICATION FOR THE PROTOCOL.....	48
E.	INTERFACE SPECIFICATION	50
F.	API SERVICES	51
1.	<i>isDeviceOK</i>	51
2.	<i>Read</i>	52
3.	<i>Write</i>	53
4.	<i>Send</i>	54
5.	<i>SharedMemoryConnection</i>	55
6.	<i>~SharedMemoryConnection</i>	56
G.	SUMMARY	57
IV.	DESIGN AND IMPLEMENTATION.....	59
A.	API ARCHITECTURE DESIGN	59
B.	API PROTOCOL DESIGN	60
1.	<i>Constructor (SharedMemoryConnection) Protocol</i>	61
2.	<i>isDeviceOK Protocol</i>	61

3.	<i>Write Protocol</i>	62
4.	<i>Send Protocol</i>	63
5.	<i>Read Protocol</i>	64
C.	IMPLEMENTATION	65
1.	<i>Constructor</i>	65
2.	<i>IsDeviceOK</i>	66
3.	<i>Read</i>	66
4.	<i>Write</i>	67
5.	<i>Send</i>	67
6.	<i>Destructor</i>	67
V.	RESULTS	71
A.	TEST ENVIRONMENT AND METHODOLOGY	71
B.	PERFORMANCE EVALUATION.....	72
1.	<i>Writing steps</i>	73
2.	<i>Reading steps</i>	73
VI.	DISCUSSION	77
A.	PROGRAMMING LANGUAGE	77
B.	PROGRAMMING NOTES	77
C.	EXTENDING INTEROPERABILITY MODEL	80
1.	<i>Core CSCI and MSG CSCI</i>	80
2.	<i>Core CSCI and Voice CSCI</i>	80
3.	<i>MSG CSCI and RF Subassembly</i>	80
4.	<i>TIO CSCI and Host</i>	81
5.	<i>Core CSCI and Terminal Exerciser</i>	82
6.	<i>TIO CSCI and Terminal Exerciser</i>	82
VII.	CONCLUSIONS & FUTURE WORK	85
A.	CONCLUSIONS	85
B.	FUTURE WORK	86
	LIST OF REFERENCES	87
	APPENDIX A. API LISTING	89
A.	C++ VERSION	89
1.	<i>ShrdMmry.h</i>	89
2.	<i>ShrdMmry.cpp</i>	90
3.	<i>MssgType.h</i>	97
4.	<i>MssgType.cpp</i>	98
5.	<i>ShmMgr.h</i>	100
6.	<i>DevcType.h</i>	102
	INITIAL DISTRIBUTION LIST	103

LIST OF FIGURES

FIGURE 2.1. TACTICAL DATA SHARED OVER LINK-16	8
FIGURE 2.2. LVT (1) SOFTWARE AND HARDWARE INTERCONNECTION	10
FIGURE 2.3. LVT (1) SOFTWARE INTERCONNECTION FOR RE-ENGINEERING	12
FIGURE 2.4. MIDS-LVT ARCHITECTURE IN CORBA	30
FIGURE 3.1. MIDS-LVT SOFTWARE STRUCTURE	36
FIGURE 3.2. TOP-LEVEL SOFTWARE ARCHITECTURE	37
FIGURE 3.3. GENERIC CSCI ARCHITECTURE	38
FIGURE 3.4. TIO CSCI ARCHITECTURE	39
FIGURE 3.5. COMMUNICATION LAYER INTERFACE	40
FIGURE 3.6. HARDWARE WRAPPER CLASS	41
FIGURE 3.7. MIDS-LVT COMMUNICATION STRUCTURE	42
FIGURE 3.8. MEMORY DEVICES CLASS	42
FIGURE 3.9. MIDS-LVT INTEROPERABILITY MODEL	43
FIGURE 3.10. MIDS-LVT RECORD PROTOCOL	45
FIGURE 3.11. CSCI INTERACTION PATTERNS	45
FIGURE 3.12. SHARED MEMORY REGION ARCHITECTURE	47
FIGURE 3.13. TIMING DIAGRAMS OF CORE AND TIO CSCI	50
FIGURE 3.14. API SERVICES FOR SHARED MEMORY	51
FIGURE 4.1. ARCHITECTURE DESIGN PATTERN	59
FIGURE 4.2. COLLABORATION DIAGRAM OF API	60
FIGURE 4.3. CONSTRUCTOR (SHAREDMEMORYCONNECTION) PROTOCOL STATECHART....	61
FIGURE 4.4. ISDEVICEOK PROTOCOL STATECHART	62
FIGURE 4.5. WRITE PROTOCOL STATECHART	63
FIGURE 4.6 SEND PROTOCOL STATECHART	63
FIGURE 4.7. READ PROTOCOL STATECHART	65
FIGURE 4.8. API UML DIAGRAM	69
FIGURE 5.1. TEST ENVIRONMENT	72
FIGURE 5.2. TIMING OVERHEAD	74
FIGURE 6.1. MSG/RF SUBASSEMBLY API	81
FIGURE 6.2. MID-STD-1553 AND ETHERNET API	81
FIGURE 6.3. CORE AND TE API	82
FIGURE 6.4. TIO AND TE API	83

LIST OF TABLES

TABLE 2.1. MIDS-LVT HARDWARE/SOFTWARE CONFIGURATION	11
TABLE 5.1 READ AND WRITE WITHOUT API.....	74
TABLE 5.2 TIME TO ACQUIRE AND RELEASE A LOCK USING API.....	75
TABLE 5.3 OVERHEAD BENCHMARKS FOR READING AND WRITING STEPS (ONE-WAY COMMUNICATION).....	75

ACKNOWLEDGMENTS

Many people have provided invaluable help during the construction of this thesis and the work it represents. I would especially like to acknowledge Dr. Valdis Berzins and Dr. Jun Ge for their guidance and support in making this thesis possible. I am grateful of their invaluable help. I would like also to acknowledge my Program Manager, Captain John Kohut and the late Mr. Kim Laporte for all their assistance in the last two years. Thanks Captain, Kim.

Special thanks to colleagues and friends that made this thesis possible: Steve Dixon, Bruce Wall and Chou Fang. Finally to my parents, my wife, and my six kids for all their support and keeping life fun.

- Theng C. Moua

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

During the arms race, software designers were under tremendous pressure to speed deployment of systems despite increasing complexity of those systems. Shipping delays were common because software designers had to manually design, implement, integrate, and test these complex software systems without the support of automated software engineering tools. Most critical deficiencies were discovered late in the integration and testing phase. With these pressures, acquisition managers routinely waived the delivery of full design documentation. In some cases, the government received the executable software without source code and documentation.

These complex systems are characterized by having heterogeneous processors connected by heterogeneous busses. Such systems required many choices regarding programming languages, development environments, and real-time operating systems, which were developed by multiple contractors to fulfill the specific requirements for these systems. Despite these complexities and challenges, the Department of Defense (DoD) has successfully developed a great number of important software systems.

Over the years, these legacy software systems have been evolving and are becoming increasingly complex while providing more functionality. The shortage of original software designs, lack of corporate knowledge and software design documentation, unsupported programming languages, and obsolete real-time operating system and development tools have become critical issues for the acquisition community. Consequently, these systems are now very costly to maintain and upgrade in order to meet current and future functional and nonfunctional requirements.

With the shrinking DoD budget, the acquisition community cannot afford to disregard these legacy systems and develop brand new substitute systems from scratch. A risk and cost reduction approach is to be developed in order to maintain and upgrade such systems effectively.

To date, the work done on the modernization of legacy, distributed embedded systems has been minimal. The majority of current methods concentrate on business and information applications. These approaches deal with the decomposition of monolithic

systems, decoupling of user interface, database management, and identifying reusable components, which may not be applicable for complex, embedded systems.

In this thesis, we focus on the re-engineering of old procedural software of the Multifunctional Information Distributed System (MIDS) Low Volume Terminal (MIDS-LVT) into a modern, object-oriented architecture in order to meet emerging requirements. The MIDS-LVT program is a joint cooperative international program that consists of five nations: United State, France, Germany, Italy, and Spain. The MIDS-LVT system is a complex distributed real-time embedded system that provides a joint and allied interoperable Link-16 tactical digital data and voice communication link among air, ground, surface, and subsurface platforms, i.e., F/A-18, F-16, EF-2000, and Patriot Missile.

The MIDS-LVT system is a product family that consists of four variations or configurations, LVT (1), LVT (2), LVT (3), and MIDS on Ship (MOS). It has eight Computer Software Configuration Items (CSCIs), which are distributed among a set of processors. These CSCIs are Core, Tailored Input/Output (TIO), Message (MSG), Tactical Air Navigation (TACAN), Subscriber Interface Army/Army Data Distribution System Interface (SIA/ADSSI), Navy Subscriber Input Output (NSIO), Fighter Data Link Input Output (FDLIO) and Voice. Depending on the configuration, these CSCIs perform parallel and serial tasks to fulfill the system functions.

During the MIDS-LVT development phase, multiple contractors from the five nations developed hardware and software components. For software applications, each contractor developed his own CSCI in an independent (a unique) language, Real-Time Operating System (RTOS), and software support environment. The implementation languages were independently chosen and different CSCI's use different languages. The Core CSCI was developed in FORTRAN and assembly languages. The TIO CSCI was developed in ADA 83 and C languages. The TACAN CSCI, MSG CSCI, and Voice CSCI were developed in C language. The SIA/ADSSI CSCI, NSIO CSCI, and FDLIO CSCI were developed in FORTRAN language. Consequently, the integration of the software subsystems was very complex.

Hardware obsolescence and inaccurate software design documentation are concerns in the MIDS-LVT program. Any change in the underlying hardware architecture is translated into a major change in all the associated software components because developers were unable to separate software functions from low-level interactions with the hardware or other software components. The details of hardware dependent communications and control mechanisms are combined with software behaviors. Therefore, any change in the hardware configuration requires significant modifications in the related software components. Without accurate software design documentation, this software and hardware interaction could become critical in our effort to re-engineer this old procedural software into a modern, object-oriented architecture.

The MIDS-LVT modernization acquisition strategy is based on a time driven, risk reduction approach. In this approach, when required, only one CSCI at a time will be redesigned into an object-oriented program while interoperability with the other unmodified CSCIs in the MIDS-LVT distributed environment will be maintained. Currently, only the Core CSCI is being considered for redesign into an object-oriented program.

This thesis proposes a new interoperability model for the MIDS-LVT system, which provides a high-level abstraction for the CSCI interfaces and interactions. Using this model, we can develop a new framework for upgrading other individual legacy CSCIs into modern software architectures.

The proposed interoperability model consists of interface, protocol, and temporal specifications. These specifications are critical for system interoperability but have not been sufficiently identified in practice. The proposed model is expected to formalize the interoperability requirements for the MIDS-LVT system and to identify and improve the component performance. After being applied in the modernization of the Core CSCI components, the model is extendable to the other CSCI components with corresponding requirement abstractions.

The interface specification consists of a set of Application Programmer's Interfaces (API), which act as the interfaces among the CSCIs' inter-processor

communication (IPC) which interact and cooperate in the MIDS-LVT distributed environment. The API is used to hide the design decisions and implementation details of how to interface with specific communication devices. The decoupling of the CSCIs' internal activities from their external application relationships allows us to understand how the CSCI components interoperate.

The protocol specification is a strict constraint mechanism or policy that controls the legal ordering of the sequence of messages involved in the interaction of two CSCIs. The use of the protocol specification provides for a safe and verifiable information exchange between the CSCIs.

For temporal specification, we are interested in the ability of the systems to schedule the functions that provide and consume the data for the interaction between two CSCIs. For CSCIs to interoperate, the temporal requirements of both CSCIs need to be compatible.

In this thesis, our research approach includes a thorough review of current real-time, distributed technology and interoperability techniques. We analyze the current MIDS-LVT requirements, interfaces, designs, test documentation, and source code to gain a thorough understanding, and then a complete abstraction of its interaction, protocol, behavior, and timing constraints. Once we gain this information, we specify, model, and design by using object-oriented, design patterns and the Unified Model Language (UML).

In our design approach, we make no assumptions about the new specific language or RTOS required for the MIDS-LVT. We will also make no assumption about the new hardware architecture in terms of a specific set of microprocessors, memory structures, data buses, and I/Os. However, our approach assumes that the MIDS-LVT hardware has sufficient CPU speed and memory space to support emerging future functional requirements.

One long-term goal in the MIDS-LVT program is to implement the components with real-time CORBA when the related technologies become mature. The intention is to develop interchangeable modules provided by competing vendors while achieving

interoperability with various platforms. The thesis should allow seamless integration with real-time CORBA technology.

This thesis is organized into the following chapters. Chapter II provides a brief overview of real-time and distributed systems, real-time operating systems, inter-processor communication, interoperability, and object-orient design. Chapter III presents the MIDS-LVT software architecture, the interoperability model, the API's specification, and the protocol specification. Chapter IV presents the design and implementation of the APIs and protocols for the MIDS-LVT. Chapter V presents the test results of the API latency time. Chapter VI presents the extension of the model for other CSCIs. Chapter VII provides the thesis conclusions and future work.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

Since the 1960s, DoD and industry have been making efforts towards economically providing sufficient computing power by using microprocessor architectures to solve complex computational application problems in fields such as strategic air defense, weather forecasting, flight simulation, telecommunication, and molecular biology. A common solution to these types of software application problems is to use parallel processing and to distribute the application over several processors. In the parallel and distributed processing environment, these types of systems exhibit substantial concurrency. Consequently, they are very complex to specify and design.

This chapter provides some background information on the MIDS-LVT system and the design challenges of migrating the MIDS-LVT into modern technology. We will also present a list of potential technologies for migrating distributed real-time systems such as the MIDS-LVT.

A. MIDS-LVT SYSTEM

The MIDS-LVT system is a complex distributed real-time embedded system that provides a joint and allied interoperable Link-16 tactical digital data and voice communication link among air, ground, surface, and subsurface platforms as shown in figure 2.1. Link-16, using the MIDS-LVT system, represents a major improvement in data link communications over the legacy data links, i.e., Link-11 and Link-4A. The following are Link-16 features:

- Jam resistance
- Security
- High data rate
- Multiple users
- Secure digital voice

- Relative navigation
- Precise participant location and identification
- A message set that supports a wide range of mission functions and data

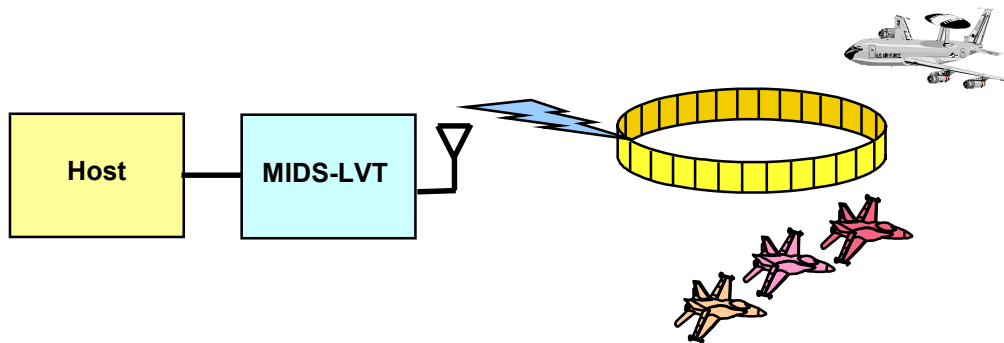


Figure 2.1. Tactical data shared over Link-16

1. System Architecture

The MIDS family architecture is defined by the terminal system segment specification, terminal interface control documents, module performance and interface specifications, and the CSCI performance and interface specifications. A module is defined as either a shop replaceable unit (SRU) configuration item or a line replaceable unit (LRU) configuration item. The MIDS hardware architecture is based on publicly available standards. It uses version E of the standard electronic modules (SEM-E). Each module in the digital subsystem is interconnected using a commercial standard Versa Module Eurocard bus (VMEbus). Each module in the radio frequency (RF) subassembly is interconnected for control and reporting using a standard RS-422 data bus.

The internal buses in the MIDS-LVT are black and red. The red bus is a VMEbus where most CSCIs interact and perform non-secure activities. The black bus is an RS-422 that dedicated for secure tasks such as signal generation and cryptography functions.

The MIDS family includes the modules and CSCIs for the LVT (1), LVT (2), LVT (3), and MOS configurations or variants, and ancillary LRUs. The main terminal unit (MTU) for the LVT (1) consists of nine SEM-E modules and three non-SEM-E modules. The SEM-E modules are:

- Data Processor (DP) / Avionics MUX (AMUX)
- Tailored Processor (TP) / Ground MUX (GMUX)
- Voice Processor (VP)
- Signal Processor / Message Processor (SMP)
- Discrete / Receiver-Transmitter Interface (RTI)
- Receiver-Synthesizer (R/S) RF / Receiver-Synthesizer digital (2 R/Ss per LVT(1) MTU)
- Exciter-CPSM / Interference Protection Features (IPF)
- TACAN Digital / TACAN RF

The non-SEM-E modules are:

- Power Amplifier (PA)- Antenna Interface Unit (AIU)
- Chassis - Harness (includes Motherboard and Front Panel)
- Internal Power Supply (IPS) (includes Battery)

The MTU of the MIDS_LVT (1) includes five operational Computer Software Configuration Items (CSCIs): Core software, Tailored I/O software, Signal/Message processor software, TACAN software, and Voice software as shown in figure 2.2. The Core CSCI provides the Link-16 message and protocol capabilities and executes on the data processor (DP) lamina. The Tailored I/O CSCI provides host-related functions and host-network communication capabilities. The host-related functions execute on the

tailored processor lamina and the host-network communication capabilities execute on the avionics MUX (AMUX) lamina and ground MUX (GMUX) lamina. The signal/message processor CSCI provides the signal generation and cryptography functions and executes on the signal/message processor (SMP) module. The TACAN CSCI provides the TACAN capability and executes on the TACAN digital lamina. The voice CSCI provides an LPC-10 and CVSD voice capability and executes on the voice processor (VP) module.

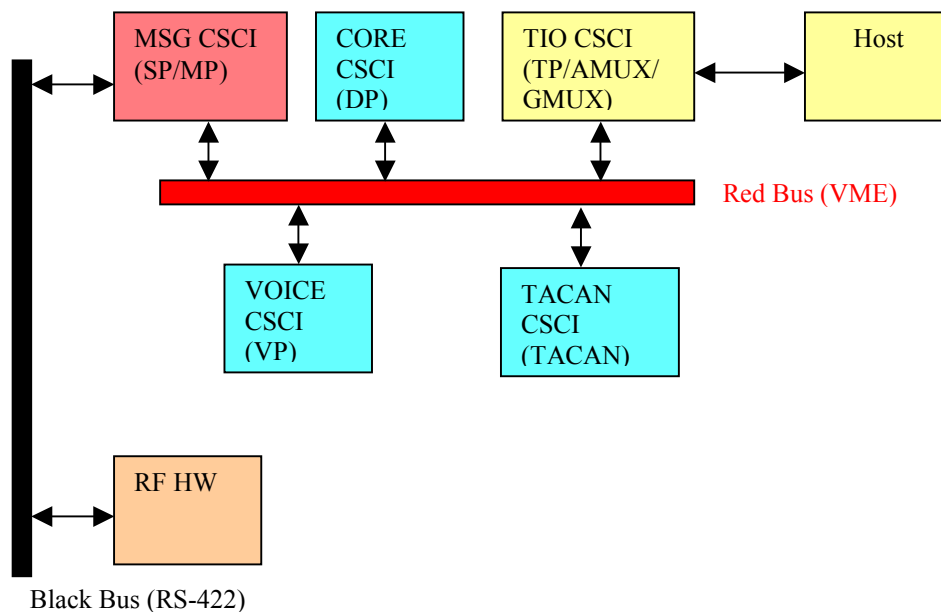


Figure 2.2. LVT (1) Software and hardware interconnection

The LVT (2), LVT (3), and MOS variants are roughly subsets of the LVT (1), with relatively few additions as shown in table 2.1. For the LVT (2) variant, the AMUX, the TP, the voice processor, one receiver-synthesizer, and the TACAN module are removed from the MTU. The tailored I/O software, the TACAN software, and the voice software are also deleted. ADDSI software is added to provide an Army-unique X.25-based host interface. This software executes on the modified-for-Army-GMUX lamina.

For the LVT (3) variant, the GMUX, the AMUX, the TP, the voice processor, the discrete, the TACAN, and the PA-AIU are removed from the MTU. A LVT (3) discrete

lamina, a LVT (3) 1553 MUX lamina, and a LVT (3) PA-AIU module are added to the MTU. The tailored I/O software, the TACAN software, and the voice software are deleted. LVT (3) interface software (FDLIO) is added and executes on the avionics MUX lamina.

For MOS variant, the GMUX, the AMUX, the TP, the discrete, the TACAN, and the PA-AIU are removed from the MTU. The tailored I/O software and the TACAN software are deleted. The MOS interface software (NSIO) is added and executes on the avionics MUX lamina.

	LVT(1)	LVT(2)	LVT(3)	MOS
DP	X	X	X	X
AMUX	X			X
LVT(3) AMUX			X	
TP	X			
GMUX	X			
VP	X			
SP/MP	X	X	X	X
Discrete	X	X		
LVT(3) Discrete			X	
RTI	X	X	X	X
R-S	X	X	X	X
Exciter	X	X	X	X
IPF	X	X	X	X
LVT(2) GMUX		X		
TACAN	X			
PA	X	X		X
LVT(3) PA			X	
IPS		X	X	
Core CSCI	X	X	X	X
TIO CSCI	X			
MSG CSCI	X	X	X	X
TACAN CSCI	X			
Voice CSCI	X			X
ADSSI/SIA CSCI		X		
NSIO CSCI				X
FDLIO CSCI			X	

Table 2.1. MIDS-LVT Hardware/software configuration

2. Re-engineering Effort

Figure 2.3 shows the Core and TIO CSCIs interconnection in our re-engineering and assessing effort. In the current MIDS-LVT legacy design, the CSCIs do not invoke other CSCIs functions directly. All the Inter-processor communications among the CSCIs are done through sending and receiving data to and from their shared memories. The current CSCIs interaction is done at very low-level. Consequently, these low-level codes were embedded within many software modules. In our re-engineering design, we propose a set of APIs which will act as interfaces to implement the CSCIs' inter-processor communication and that will hide the design decisions and implementation detail of how to interface with specific communication devices. The new re-engineering designed CSCI should act and feel the same to other unmodified legacy CSCIs.

In our current modernization acquisition strategy, the Core is the first CSCI that will be redesign into an object-oriented architecture. The Core CSCI was inherited from Joint Tactical Information Distribute System (JTIDS) and was developed in Fortran and assembly language beginning in the early 1980s. After the many years of software changes, the current software architecture of Core CSCI is very fragile and unable to accommodate new complex requirements. Simple changes in the Core CSCI may require large software development efforts and extensive regression testing which is costly for the users.

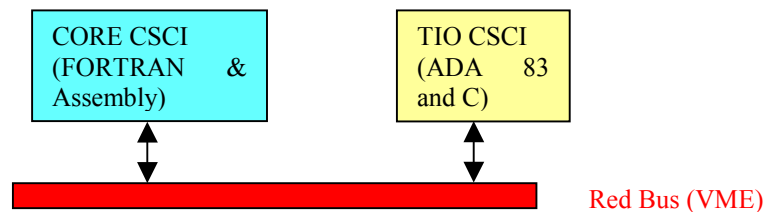


Figure 2.3. LVT (1) Software interconnection for re-engineering

B. DISTRIBUTED REAL-TIME SYSTEMS

Designing software for distributed real-time systems is very complex due to several aspects of the distributed and real-time characteristics that are not applicable non-distributed real-time systems. Specifically, distributed real-time systems must implement true concurrency, which means that they must support the simultaneous execution of several high-level tasks. They are extremely complex to specify and develop because many interdependent operations must execute on different processors at the same time. When the system is large and consist of several variations, the interactions resulting from simultaneous operations make it very difficult for developers to understand the implications of their design decisions.

This section provides an overview of real-time systems, distributed systems, and RTOS.

1. Real-Time System

A real-time system (RTS) is a concurrent system that has performance deadlines on its computations and actions [Ref. 2]. Real-time systems have wide spread use in military, industrial, and commercial applications. A RTS usually consists of a RTOS, I/O, and several sensors and actuators. RTS is classified as hard or software. A hard RTS has time-critical deadlines that must be met. In a soft RTS, missing the deadline is not desirable but it will not cause a mission failure if it does not occur too often.

RTS applications software is designed to operate in a real-world environment. Its have several characteristics that distinguish them from other software systems. The MIDS-LVT has the following characteristics:

a. Embedded System

A RTS can be an embedded system. In this case, the RTS is a component of a larger hardware/software system. Often there is no direct human interface. The RTS only responds to external stimulus via sensors and transducers. For example, the MIDS-

LVT is part of the host platform (F/A-18, F-16, and F-15) communication and navigation subsystem.

b. Timing Constraint

A RTS has timing constraints. For example, the MIDS-LVT must process events within given time slots. Failure to complete the task might be catastrophic for the system.

c. Real-Time Control

A RTS often involve real-time control. It makes decisions and produces control signals based on input data without any human intervention. For example, the MIDS-LVT synchronizes to the network by adjusting its internal time base to the data received from the host and network.

d. Time-Driven System

A RTS is a time-driven system. Its actions are driven primarily by periodic tasking or the arrival of time epochs rather than by the arrival of aperiodic events. If tasks are not complete by the time of the arrival of the next time epoch then the system fails to meet its time requirement. This is not acceptable in mission critical systems. The MIDS-LVT must process all of its data before the arrival of the next time epoch.

e. Reactive System

Many RTS are reactive systems. They are event driven and must respond to external events. For example, when the system is driven by the occurrence of external events (clock alarms, signals) the system must react to these events. The system does not read its input or control when such inputs occur. It simply reacts to their occurrence.

2. Distributed System

Distributed systems can range from small and simple to large and complex, usually running on separate computers that are in geographically different locations [Ref. 3]. For example, a program on one machine that is at distant location is able to interact

with a different program on a second machine through an underlying data communications mechanism. In large and complex distributed embedded systems such as the MIDS-LVT, the programs dispersed over a set of distributed processors that connected by internal global busses.

There are two models for distributed systems: asynchronous and synchronous.

a. Asynchronous Model

An asynchronous distributed system model consists of a set of processors that run at their own speeds, do not share a common clock, do not have synchronized clocks, and do not share any memory. All communication among the processors occurs by message-passing and there is no predictable upper bound on the time it takes for the communication network to deliver a message. Predicting the latency of the network and the resource on a distributed system that you do not control can be very difficult. Due to the latency issues, an asynchronous distributed model is not used for designing hard real-time system.

b. Synchronous Model

A synchronous distributed model assumes that the upper bounds on the communication delay and computation latency are always known. This system consists of a set of processors that share a common global clock that synchronizes and coordinates the common tasks among the different processors to meet the strict timing constraint required by the system.

Synchronous distributed application is a concurrent application. It may execute in an environment consisting of multiple heterogeneous processors that are connected together by heterogeneous busses. The MIDS-LVT is a system of this type.

3. Real-Time Operating System

A Real-Time Operating System (RTOS) provides special services for real-time programming applications [Ref. 2]. These special services include: rapid response within known bounds to external and internal events, interrupt handling, task scheduling and

dispatching, response to timer events, and provision for mutual exclusion. Examples of typical RTOSs are INTERGRITY (Green Hills Software), QNX (QNX Software Systems), RT-March (CMU), RTMX O/S (Open BSD + Real-Time extensions), Solaris (Sun), Spring Kernel (U. of Massachusetts), VRTX (Mentor Graphics), VxWorks (Wind River Systems), and many others.

C. BENEFITS OF REAL-TIME DISTRIBUTED APPLICATIONS

Distributed processing has several advantages and benefits over a single processor environment.

1. Load Balancing

In a mission critical system, load balancing is crucial for the success of the system operation. In distributed environment, the overall system load can be shared among several CPUs.

2. Improved Response Time

If the system has many external requests, these requests can be assigned to different processes working concurrently. In this case, the external requests can be processed in a more timely fashion.

3. More Economical

With the rapidly declining costs and rapidly increasing performance of microprocessors, a distributed solution is cheaper than a centralized solution.

4. Improve Scalability

A given application is scalable. It can be configured in different ways by selecting an appropriate number of CPUs to support the requirements.

5. Extendable

If the system requires more performance, it can be extended by adding more CPUs to prevent system overload. In this case, new, additional functionality can be allocated over the extended set of processors.

D. CHALLENGES OF DEVELOPING REAL-TIME DISTRIBUTED APPLICATIONS

Despite the advantages and benefits provided by a distributed environment, developing this type of application is non-trivial task. There are many different design trends associated with the possible solutions, but no current solution is able to resolve all the problems and issues.

In the following section we will discuss the challenges relates to implementing a RTS.

1. Concurrency and Schedulability

Dijkstra recognized the growing number of applications, including that real-time and distributed systems were concurrent in nature, in which several activities were logically occurring in parallel [Ref. 15]. The issues of concurrency and schedulability are strongly related. Concurrency is the simultaneous execution of multiple sequential chains of actions. Schedulability is the selection of task to execute next from among all tasks that are capable of executing. Schedulability analysis tries to determine whether a system composed of many tasks can meet its entire deadline. For instant, a single processor system can do only one single thing at a time and, therefore, it must implement a

scheduling policy that controls when the different tasks execute. For a large and complex application, execution on single processor system may too slow and, thus, not sufficient to support hard real-time requirements. The application that takes a long time to run may be speeded up by dividing the work of the application among separate processes that can run concurrently on different processors. Designing an application in a distributed environment required careful partitioning, coordinating, and scheduling tasks on different processors to achieve true concurrency.

Uncontrolled concurrency can be dangerous. For example, one process reads an object while the object is being written concurrently by another process. The first process might see the object while it is in a temporarily inconsistent state and might fail for this reason. To guard against such problems, software designers normally use synchronization mechanisms, which are primitive operations provided by the RTOS to ensure the correct synchronization of the processes. In a heterogeneous distributed environment such as the MIDS-LVT where the system consists of many types of RTOSs, the synchronization mechanism provide by the specific RTOS may not interoperable with the other RTOSs.

2. Timing Constraint

A RTS depends not only on the logical results of the computations, but also on the times at which those results are produced. For a hard RTS, it must produce functional results by a specific deadline. Otherwise there may be catastrophic consequences for both the system and the environment it operates in. Often it is impossible to predict with when particular events will occur, what their order of occurrence will be, and how long they will last. For the MIDS-LVT, all tasks must be completed before the end of each time slot, which is 7.8125 millisecond. Missing this deadline may be critical depending on type of data that it's processing. For example, if the unprocessed incoming data of the MIDS-LVT are related to navigation, by missing this deadline, the host may not be able to correlate the MIDS-LVT navigation solution with others avionics sensors data. This

will affect the host's ability to perform its mission and its safety. To guard against this condition, each CSCI satisfy its own timing constraints before the end of slot interrupt.

In general, timing constraints must be expressed, enforced, and their violations handled [Ref. 2]. Commonly, the timing constraint expression can take the form of start times, deadlines, and periods for activities. The timing constraints are related to execution time and its enforcement requires predictable bounds on activities.

All computing systems usually share resources serially to achieve a required function. Therefore, designing of real-time systems should be concerned with specific timing, scheduling, and execution ordering constraints that all processors must obey.

3. Dynamic Behavior

An important aspect of many RTSs is their dynamic behavior during run-time. The dynamic behavior of a system must be predictable. This is crucial for many safe-critical and high-reliability systems, such as avionics systems (MIDS_LVT), medical systems, and nuclear power plants.

4. Correctness and Robustness

A system is correct when it does the right thing all the time [Ref. 2]. If the system does all the right things under both planned and unplanned circumstances then such a system is robust. These are non-functional real-time requirements that systems such as the MIDS-LVT must satisfy.

E. INTER-PROCESSOR COMMUNICATION

In a homogeneous RTOS environment, the inter-processor communication (IPC) mechanisms for exchanging data elements between processes that reside on different processors can be easily implemented using the underlying RTOS services. IPC

mechanisms can be implemented in various ways, including message passing, shared memory, and signaling.

In distributed environment, the system normally consists of several heterogeneous RTOSs. In this case, the underlying RTOS does not support an IPC mechanism across its RTOS boundary. For example, if program A using RTOS A' and program B using RTOS B' are to interact. These two programs cannot use their underlying RTOS mechanisms directly to communicate to each other. For instant, shared memory or mutual exclusion semaphores that are created by RTOS A' will not be accessible from program B using RTOS B'.

In this section we discuss different IPC techniques used in homogenous RTOS distributed embedded environment. Our goal is to exploit the features of these IPC techniques for heterogeneous environments such as the MIDS-LVT.

1. Shared Memory

The shared memory is an unbuffered communication technique. The unbuffered data is accessed through shared memory, which need mutual exclusion (locks) to read from and written to. Shared memory enables multiple processors to share a data area and to transfer data among themselves. The shared memory can be implemented in several ways depending on the RTOSs and requirements of the applications.

In the MIDS-LVT, the shared memory is a physical memory that the processors can access. The hardware does not provide the locks and synchronization for accessing the shared memory. The locks must be implemented in software. Our shared memory data structure consists of a data transfer block area, a pointer area, and a handshake area. The processors using the shared memory must determine and provide restrictions as to the content, organization, and usage of the data area in the shared memory. The processors must also synchronize the use of a shared memory. Consequently, thoughtful programming, usually involving events or signals, is required to enable several processors to update a shared memory.

The shared memory is the most common form of and perhaps the fastest inter-processor communication mechanism, especially for transferring large structures between multiple CSCIs of hard real-time systems. However, they require careful synchronization or subtle bugs can occur in the complex software.

2. Pipe

The pipe is a buffered communication technique, which allows processes to communicate. A Pipe is a first-in first-out (FIFO) buffer, which enables concurrently executing processes to communicate data: the output of one process (the writer) is read as input by the second process (the reader). Communication through pipes eliminates the need for an intermediate file to hold the data.

In most RTOSs, pipe is a shared memory that is unnamed. Pipes are used to send and receive data between two processes in the processor. Pipe data may arrive at any time. When used in hard real-time systems, the designer must determine upper bounds on the number of produced and consumed messages to enable guarantee of temporal properties. Pipes may not operate in a heterogeneous operating system environment.

3. Distributed Shared Memory

Distributed shared memory (DSM) provides transparent reads and writes of shared data in a networked environment [Ref. 6]. The functionalities of a DSM system are built to provide an illusion of a global virtual memory and to support concurrent writes on different nodes. For MIDS-LVT to work correctly using this technology, the CSCIs or at least the parts responsible for interaction must be implemented using the DSM.

4. Signals

In inter-process communications, a signal is an intentional disturbance in a system. The signals are designed to synchronize concurrent processes, but they can also

be used to transfer small amounts of data. Signals are usually processed immediately and provide real-time communication between processes.

Signals are also referred to as software interrupts because a process receives a signal similarly to how a CPU receives an interrupt. Signals enable a process to send a “numbered interrupt” to another process. If the active process receives a signal, the intercept routine is executed immediately and then the process resumes execution where it left off.

In the MIDS-LVT, we use hardware signals such as EOS and DTI to synchronize the CSCIs that reside on different processors. Normally the signal’s mechanism that is provided by the RTOS cannot operate across heterogeneous operating system boundaries.

5. Alarms

Alarms enable programs to send signals or to execute subroutines at specific times or at specific intervals. The program can arrange for the signal to be sent at a specific time of the day, after a specific interval has passed, or sent periodically.

A cyclic alarm is most useful for providing a time base within a program. This greatly simplifies the synchronization of certain time-dependent tasks. For example, a real-time simulation might allow one second for an instrument to update. A cyclic alarm signal could be used to determine when to update the display.

The advantage of using cyclic alarms is more apparent when multiple time bases are required. Each function could be given a cyclic time to process the data. The alarm can be used for external control and to synchronize the CSCIs’ communication.

6. Mutual Exclusion

In concurrent systems, more than one process might want to access the same resource simultaneously. For example, if two processes need to communicate with each other through a common shared memory, it may be necessary to synchronize the processes so that only one updates the shared memory at a time. Semaphore is a

mechanism that is used to synchronize concurrent processes that are accessing critical section is mutually exclusive. Dijkstra provided a solution to the classical mutual exclusion problem by using binary semaphore [Ref. 15]. The binary semaphore has two values: zero and one. When the semaphore is set to one, it means that the resource is free. When it is decremented to zero, the resource is already acquired by other task. In certain applications, we may want more than one process to read the shared resource, providing that no more than one process is writing into the critical section at the same time.

F. OBJECT-ORIENTED METHODOLOGY

As mention earlier, real-time design is a complex process, primarily because of the added constraints that must be met, i.e., temporal, resource, load balance, scheduling, and inter-processor communication. Because of these constraints, the current practice for building a successful RTS often involves art as much as it does science [Ref. 5]. In improving the effectiveness of designing real-time system, Object-Oriented (OO) has the potential and it is becoming a popular option.

Object-orientation is a software development paradigm that allows the engineer to view and model the world as a set of interacting objects. The promise benefits of OO are software reuse, improved system partitioning, and clearly specified interfaces.

While OO has been successful in designing commercial software, what is not well understood is how the technology can be best applied to large complex real-time systems. We highlight a few of the essential concepts underlying OO analysis and design techniques for real-time systems.

1. Abstraction

Abstraction focuses on the essential aspects of an entity and ignores or conceals less important or non-essential aspects. An OO approach encourages the construction of abstractions, both of the real-world of the system and of the problem. It is a fundamental tool in handling the complexity of large software systems.

2. Encapsulation/Information Hiding

An OO technique provides a more obvious and natural mechanism to limit access to shared data by encapsulating or hiding information. OO provides a single construct called a class that encloses both data and functionality. It only exports a necessary subset as its public interface, keeping the rest private. This approach minimizes the impact of requirement changes and reduces the risk of abusive implementation. It also reduces the complexity of developing large software systems.

3. Inheritance

Inheritance is a mechanism provided by OO approaches to enable class refinement and reuse. Reuse is achieved by inheriting data and functions from one class into another.

4. Polymorphism/Dynamic Binding

The ability of a real-time system to behave within a predictable and consistent tolerance range is often the foundation for success. With the exception of Ada95, polymorphism and dynamic binding impose a predictability problem in some object-oriented language (C++ and Java), since binding can take different amounts of time for different objects due to object hierarchy and overheads. Most real-time systems resort to early or static binding, trading improved predictability for lower flexibility. This is a compromise made in order to ensure that the real-time constraint can be met.

G. DESIGN PATTERNS

A design pattern is a generalized solution for a commonly occurring type of problem [Ref. 7]. A pattern permits the reuse of a successful design. Each pattern describes a problem, which occurs repeatedly in the environment, and then describes the

core of the solution to that problem. A pattern usually does not give the detailed information for a particular solution. The user of the pattern must adapt the pattern to a particular case at hand and supply the missing details not given the pattern.

Design patterns can be recognized at many levels of scale and in many disciplines. In computer science, large-scale patterns usually used to represent architecture or models and small-scale patterns represent common arrangements of programming language constructs. By means of design patterns knowledge of good software design can be documented and the experience gained within software projects widely distributed. With design patterns, a common design vocabulary is introduced, simplifying communication between software engineers.

Four design patterns, façade pattern, decorator pattern, strategy pattern, and proxy pattern were found interesting for designing our API for the MIDS-LVT.

1. Façade Pattern

A facade pattern provides the interface to the object. It defines a higher-level interface that make the object easier to use, i.e., abstract out the complex detail implementation of that object. This pattern provides layer support so we can define the API as an entry point to the shared memory for each CSCI.

2. Decorator Pattern

A decorator pattern is the same as wrapper pattern. It encloses an object of one class in another class that “decorates” the original objects (as a border around a window). It can be used to adapt an existing API to fit another API specification.

3. Strategy Pattern

A strategy pattern defines a family of algorithms. It encapsulates each one and makes them interchangeable. This pattern allows the algorithms to vary independently

from the clients that use them. We can use this pattern to represent the communication protocol. The protocol policy can be extended for a child without modifying the parent's protocol algorithm. It provides one size fits all interfaces without forcing a one algorithm to fits all implementation.

4. Proxy Pattern

A proxy pattern provides a placeholder for another object to control the access to it. This pattern is useful to representing device I/Os such as serial, Ethernet, and MIL-STD-1553 communication of the MIDS-LVT.

H. GENERATIVE PROGRAMMING

An interesting emerging approach that has the potential to deal with designing complex families of RTSs is generative programming [Ref. 1]. Generative Programming (GP) is a new software engineering paradigm that focuses on modeling a family of products rather than a one-of-a-kind systems. GP techniques enable the automated generation of a product from existing components with a given requirement specification. The GP approach is based on the generative domain model, which consist of three elements:

- Specifying family members, i.e., systems for specifying the MIDS-LVT
- Implementation components, i.e., MIDS-LVT components that can be assembled
- Configuration Knowledge, i.e., the knowledge of assembling the MIDS-LVT based on the specification

In GP, feature modeling is used to capture important feature and variation points that are easily missed as the basis for deriving the implementation components for the system family. This notation has many advantages over notation such as the Unified Modeling Language (UML).

Czarnecki and Eisennecker [Ref. 1] recognized the importance of capturing production knowledge of the software systems. They point out that having only the production software for a system without the design knowledge and an understanding of the specific design process used has contributed to the difficulty and high cost of software evolution and maintenance. This is true in every major military software application.

I. CORBA TECHNOLOGY

Common Object Request Broker Architecture (CORBA) [Ref. 9] is an open systems standard developed by the Object Management Group, which allows communication between objects on heterogeneous platforms. It is the de facto standard for integrating and deploying distributed applications in heterogeneous computing environments.

1. Properties

CORBA has three key properties that allow systems to achieve interoperability among multiple vendors.

a. Heterogeneous Environment

CORBA is designed for platform and operating system independence. Today, well over 50 different operating systems support CORBA.

b. Language Independent

CORBA is designed for language independence. CSCIs implemented in one programming language can communicate transparently with other CSCIs implemented in different languages. CORBA interfaces are standard for C++, Java, Ada, C, COBOL, Smalltalk, and Lisp.

c. Location Independent

CORBA applications are location independent. CSCIs do not need to know each other's physical location on the network or bus.

2. Object Request Broker

Object Request Broker (ORB) is the essential element of the CORBA technology. The ORB is a middleware or software bus that sits between a distributed application and the underlying communications transport layer. Distributed objects are accessed through the ORB. The ORB is responsible for tracking the objects' locations and managing all communications with an object. The ORB has the capability to resolve the incompatibilities that may exist between two systems' native data representation. The most important feature of an ORB is its ability and responsibility in selecting the communication channel, including communication over shared memory in the same processor, across a backplane if multiple processors are part of the same system, or using TCP/IP across a local-area network. When TCP/IP is used to access a remote object, ORBs communicate with each other using the Internet Inter-ORB Protocol (IIOP) standard. The use of IIOP ensures interoperability between different vendors' ORBs.

3. Interface Definition Language

CORBA achieves programming language independence by employing a language-independent interface definition language (IDL). The IDL specifies the interface to distributed objects. An IDL compiler simplifies application development by generating source code stubs and skeletons that make remote object invocation appear local. An application invokes the CORBA object by calling the client stub. Likewise, the skeleton provides a native language wrapper for the servant code that implements a distributed object.

CORBA can integrate legacy application components by defining an IDL that corresponds to its interface. The wrapper code can then be provided to map between the skeleton generated by the IDL compiler and the legacy interface. The main benefit of this approach is that any language for which an ORB is available can utilize the IDL interface. For example, new software for the MIDS-LVT written in C++ can easily access code written in FORTRAN.

4. Real-Time/Minimum CORBA

Recognizing the potential use of CORBA for connecting and integrating embedded applications, the OMG has produced two specifications. These two specifications were specific for embedded and real-time systems: Minimum CORBA and Real-Time CORBA.

a. Minimum CORBA

The Minimum CORBA standard defines a subset sufficient for most embedded applications and well suited for resource-constrained environments. The omitted features represent a trade-off between usability and conserving resources. This new specification is design for small-embedded systems, i.e., TV, microwave oven.

b. Real-Time CORBA

The Real-Time CORBA specification extends CORBA so that it can be used to build predictable real-time distributed systems. Obtaining this predictability typically requires that all the CSCI components behave predictably. This is a prerequisite for ensuring real-time performance. For Real-Time CORBA to be successful in real-time systems, its behavior must be predictable.

Real-Time CORBA addresses this by providing mechanisms to control the use of the processor, memory, and network resources. Specifically CORBA:

- Allows mapping of priorities and scheduling down to the underlying RTOS tasks/threads.
- Allows controlling the amount of memory resources to be used in a predictable fashion.
- Allows the application to select between available and make choice about the amount of sharing of the connections that occurs.

After reviewing several studies on this technology, we concluded that real-time CORBA still not mature enough for use in a hard real-time system [Ref. 10] [Ref. 11] [Ref. 12]. The main concerns are its large footprint, performance, predictability and reliability. One long-term goal in the MIDS-LVT program is to implement the components with real-time CORBA when the related technologies become mature.

In figure 2.4, we show how the MIDS-LVT would map into CORBA technology by replacing our API with CORBA IDL. The external security and host applications will require adapters or wrappers to communicate with the MIDS-LVT.

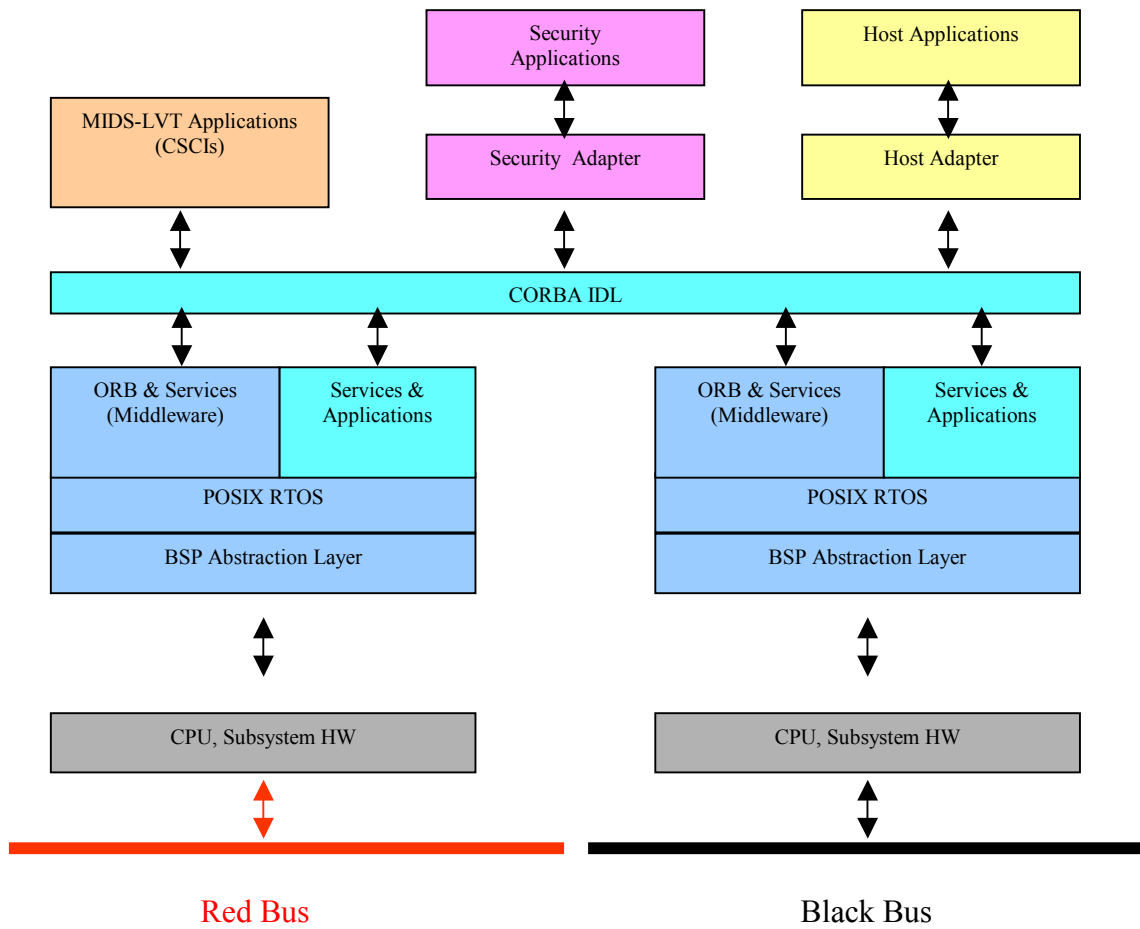


Figure 2.4. MIDS-LVT architecture in CORBA

J. INTEROPERABILITY TECHNIQUES

Interoperability concerns the ability of two or more systems or software components to communicate and cooperate with each other. The interoperability problems can arise from many situations i.e., the integration of a new system with legacy systems or the reuse of legacy software components that need to be connected in order to satisfy new requirements. We can view interoperability at different levels of abstraction. A component is a lower level while a system is the highest level of abstraction.

Abstraction refers to what parts of the program's structure and behavior are hidden and what parts are visible. The highest level of abstraction is the model of the behavior of the whole system, which includes no structure information. A lower level of abstraction is the model in which the structure of the modules is hidden and only the interaction of the modules is visible. Below this level, there is the model of the code of individual units. An even lower-level model makes the machine code visible, but that is not very useful for purposes of determining interoperability.

Interoperability problems arise not only in a homogeneous environment, but also in a heterogeneous environment. For instant, we may want a CSCI that written in language A with RTOS B to interact with a CSCI written in language C with RTOS D. In this case, problems may occur at both syntactic and semantic levels. That is, the two CSCI may compile without an error but the functionality of the interoperation may not be what is expected.

A software component takes on many meaning depending on the people involved in the software development environment. In general, components could be functions, objects, or subsystems or software modules that consists of multiple functions. In this thesis, we use the term software component as an implementation software unit of an object-oriented or procedural language and which can be composed with other units [Ref. 1]. Our CSCI is composes of many software components and can be further composed with other CSCIs to form a system.

To understand interoperability, we reviewed the existing techniques for determining software interoperability including Zaremski and Wing's specification

matching [Ref. 13], Polyolith system [Ref. 14], CORBA IDL [Ref. 9], Software Adaptor, Interoperable Component Model (ICM) [Ref. 4], and Object-Oriented Model for Interoperability (OOMI) [Ref. 8]. Each technique provides features that are unique in what it requires to achieve interoperability. Some techniques consider only syntactic level interoperability while others consider both syntactic and semantic level of interoperability. Among all the techniques that we reviewed, CORBA IDL was the only technique that supports both procedural and object-oriented language. Similar to other middleware techniques, CORBA IDL supports interoperability only at the signature level. With the exception of the OOMI, all methods are fine-grained approaches for defining components and their interactions. One area that none of the techniques addressed is the temporal requirement, which is an important feature that is required by all hard real-time systems.

Our goal is to exploit the features provide by these various research efforts and to construct a new interoperability model for modernizing the MIDS-LVT. Our model shows the CSCIs' interaction in a distributed embedded environment. Our model addresses the interoperability problem at coarser granularity than the component level.

In our interoperability model, each CSCI consists of interface, protocol, and temporal specifications. The interface is comprised of methods (API), which acts as the interfaces for CSCIs to interact and cooperate in the MIDS-LVT distributed environment. The protocol is the sequence of messages involved in the interactions that occur between the CSCIs. The temporal is the timing requirement and constraint in these interactions. In chapter three of the thesis, we describe in detail our interoperability model.

F. SUMMARY

In this chapter, we introduced the MIDS-LVT system and its design challenges and presented a list of technologies that the MIDS-LVT can use in its modernization to achieve its interoperability and maintainability goal.

One important message we would like to point out is that developing portable, reusable, and efficient distributed real-time embedded software is not a trivial task.

Many reasons contribute to this complexity including heterogeneity, communication and computation latency, synchronization, coordination, concurrency, and schedulability of common tasks to achieve system requirements.

CORBA is emerging as a promising tool in the distributed real-time embedded environment. The benefits promised by CORBA (abstraction, heterogeneity, etc.) are appealing in many application domains, including real-time embedded system such as the MIDS-LVT. Unfortunately, CORBA was not designed for real-time distributed applications. The performance, predictability and reliability of current available ORBs are still not mature for use in hard real-time systems such as the MIDS-LVT. Therefore, new ORBs still need to be designed, implemented, and tested before the MIDS-LVT can use it.

Current interoperability techniques and models do not fit our modernization approach for the MIDS-LVT. In our approach, only one CSCI at a time will be redesigned into an object-oriented program while interoperability with other unmodified CSCIs in the MIDS-LVT distributed environment will be maintained. None of the current interoperability models addresses our problems.

For the purpose of this thesis (limited in this thesis), we are not addressing all the issues that have been identified in this chapter. We propose an interoperability model that will allow us to migrate one MIDS-LVT CSCI at a time into an object-oriented program while maintaining interoperability with the unmodified legacy CSCIs. This is accomplished through using the API, protocol, and temporal specifications. The API will allow us to separate the CSCI's internal activity from its external relationships. The protocol will provide us with a strict constraint mechanism and policy to control the legal ordering of the sequence of messages involved in the interaction of two CSCIs. The temporal specification will provide us with the timing requirements and constraints for the interactions of the MIDS-LVT CSCIs.

In Chapter III, we will present the specification and the design of the API.

THIS PAGE INTENTIONALLY LEFT BLANK

III. SPECIFICATION AND MODEL

This section gives an overview of the new MIDS-LVT software architecture, interoperability model, and describes its features and underlying design.

A. MIDS-LVT SOFTWARE ARCHITECTURE

When designing the new MIDS-LVT software architecture, we followed a sound software engineering principle, which is to decompose the software into multiple layers so that systems can be reused and easily deployed. We used layers to facilitate component-based software. Software components can reside in different logical layers and be separated by reliable interfaces. Components adhering to the appropriate interface can easily be assigned to any given layer. That is, the top layer does not send messages to the bottom layer, and vice versa. Instead, the top layer sends messages to the adjacent layer and the adjacent layer sends to the next layer until the messages reach the bottom layer. For distributed systems, the use of layers is important to the overall operation and flexibility of the system as it allows components to be physically dispersed across a set of processors.

The structure of the architecture framework is shown in figure 3.1. We will show how the MIDS-LVT is mapped into the different layers or tiers. Our architecture framework consists of five layers. Layer number one represents the hardware of the system, i.e., CPUs, buses, and devices. The second layer is the board support product abstraction, which includes device drivers and unique libraries required for the devices. The third layer is the RTOS and its facilities. The fourth layer is our API for the MIDS-LVT application. The top layer is our system application software. This is the highest level of abstraction.

The key benefits of this software architecture are:

- It maximizes the use of commercial products.
- It isolates the domain application from the underlying hardware through multiple layers.

- It provides for a distributed processing environment through the use of API to provide software application portability, reusability, and scalability.

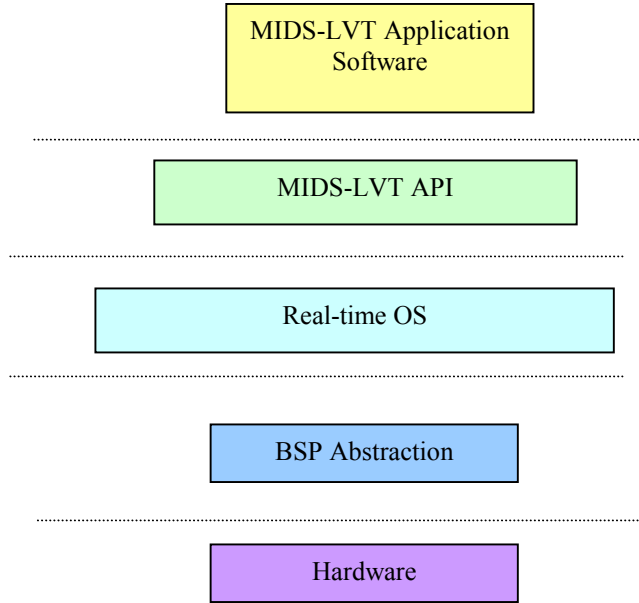


Figure 3.1. MIDS-LVT software structure

In figure 3.2, we show the top-level software architecture view of the MIDS-LVT. The software architecture represents a high-level abstraction of the system. The UML diagram shows five CSCIs that reside on the MIDS-LVT (1) and a hardware wrapper class. Each CSCI class represents a large concept in the application domain. The CSCI is not an object nor a function but a package of classes, associations, operations, events, and constraints that are interrelated and have a well defined interface specification. In the MIDS-LVT (1), these CSCIs are the Core, the TIO, the MSG, the Tacan, and the Voice. Each CSCI consists of many software components. A component is a software unit with sufficient specification for composition and interoperation with other components.

The Hardware Wrapper class encapsulates the real-world hardware devices. It is defined as an abstract class, with no direct instances. This abstract class contains interfaces that may be replaced or extended by a specific concrete class. These interfaces

shield the designer from the internal complexity of the real-world hardware devices, i.e., shared memory, the Ethernet, the MIL-STD-1553 Bus and the RS-422.

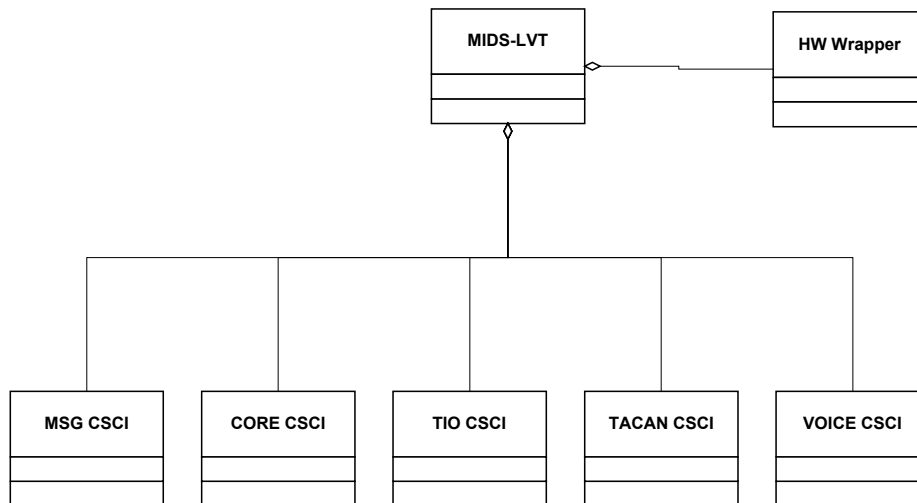


Figure 3.2. Top-Level software architecture

1. CSCI Architecture View

Figure 3.3 shows the CSCI class architecture that consists of aggregation and inheritance features for several specific classes. Our approach here is to provide the parent class with the necessary methods for management and control while allowing the child classes to share the methods defined by their parents. For example, when an exception is to occur in one of the child classes, the child class may not have a handler if it is not defined. In our design, the child class always has the ability to share exception-handling mechanisms inherited from the parent class. The child component class will inherit all basic methods and attributes from the parent class. As many child components as needed should be defined to meet the CSCI requirements. One important point is that we want the parent class to have the ability to control and manage the child classes. The class data messages should be designed into the abstract data type (ADT) to better support the modification. This is an important feature for object-oriented programs because the ADT hides implementation details.

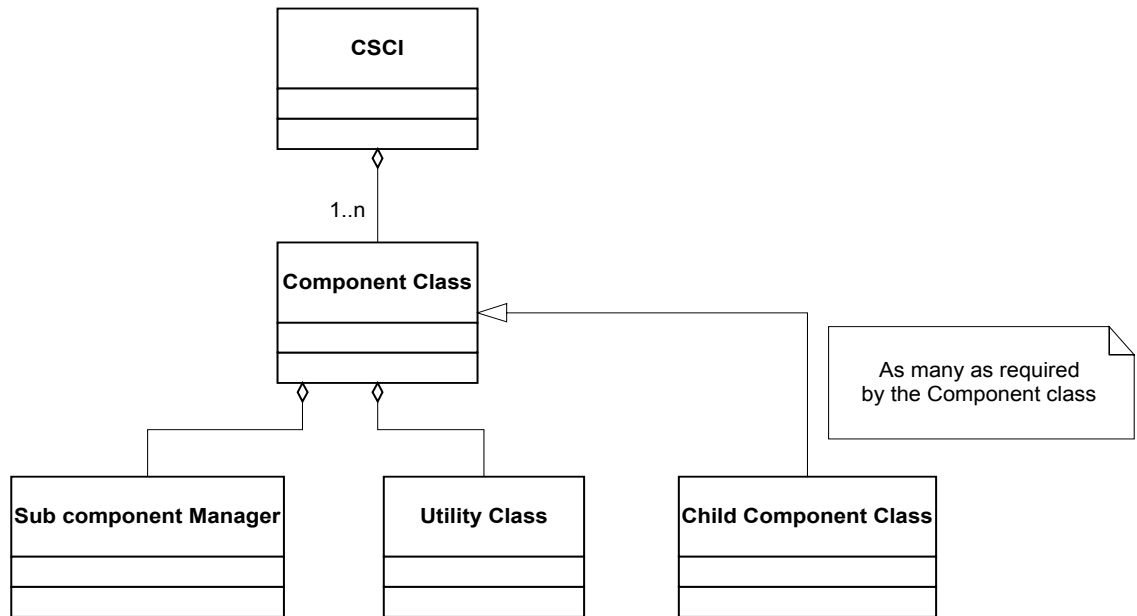


Figure 3.3. Generic CSCI architecture

In figure 3.4, we show a high-level view example of the TIO CSCI mapping with our generic CSCI architecture. The TIO CSCI consists of three major CSCs: the Tailor Process (TP), the Avionics Control Process (ACP), and the Ground Control Process (GCP). The TP CSC is the base class for communications, built-in-test (BIT), navigation, boot, etc. Specifically, it defines the exception-handling mechanisms for all child classes. The ACP CSC is responsible for filtering messages and communicating with hosts that communicate via MIL-STD-1553 and 3910 buses. The GCP CSC is responsible for filtering messages and communicating with hosts that communicate via Ethernet and X.25. In this example, we show one class - *TPFilter*. The *TPFilter* class converts data between various hosts' navigation formats and the Core CSCI's navigation data format. This class inherited exception-handling mechanisms from the parent *TP* class. The *TPFilter* class is the parent for two others classes – *TPFilterBIT* and *TPFilterNavA*. *TPFilterBIT* class filters BIT data before sending it to the host. *TPfilterNavA* class filters navigation data for platform A before sending it to the host.

Other classes can be extended from the *TP CSC class* to support the TIO CSCI requirements. The *TPManager class* provides control and management functions for the TP CSC. The *TPUtilities* class is a collection of free subprograms. In C++, *TPUtilities* are classes that only provide static members and static methods.

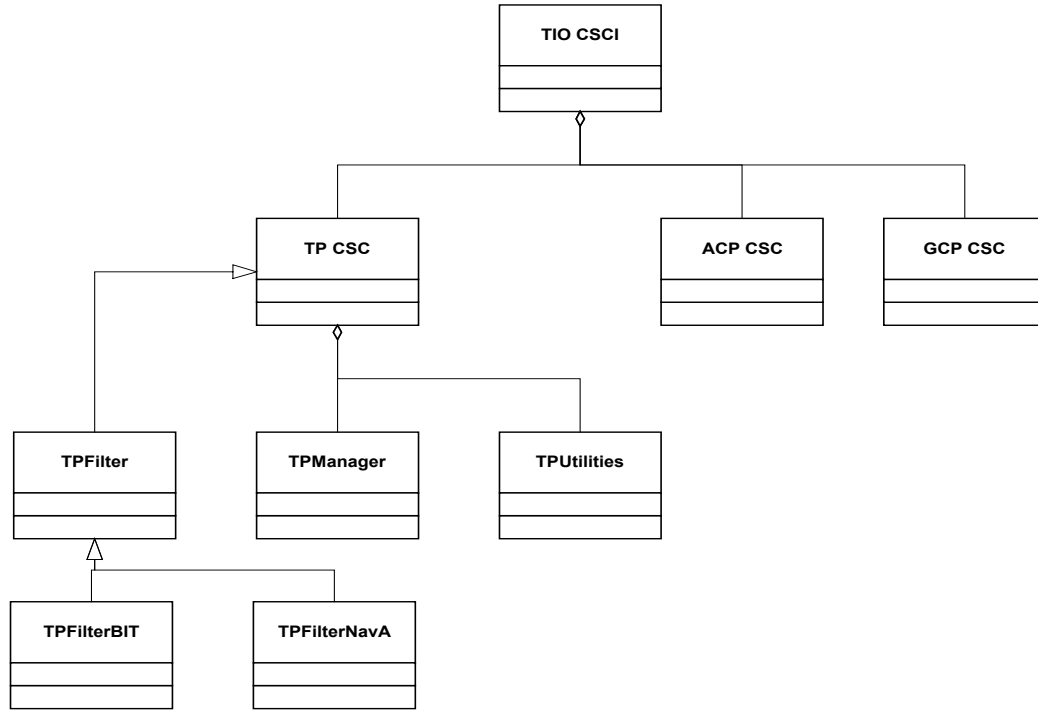


Figure 3.4. TIO CSCI architecture

2. Hardware Wrapper

The Hardware Wrapper class encapsulates the real-world hardware devices that exist in MIDS-LVT system. We use the information hiding principle to hide the design decision of how to interface to the specific I/O device. Our approach is to provide a virtual communication interface layer in order to hide the device-specific implementation details. If the software designer decides to replace the hardware device with a different device, which has the same overall functionality, the content of the communication object

will need to change. However, the virtual communication interface representing the operation as shown in figure 3.5 remains the same.

In the case of a shared memory, the communication device driver interface layer is not necessary since we can access the physical memory from the virtual communication interface directly without going through the specific device driver. For instance, every commercial MIL-STD-1553 Bus device has its own device driver, which represents the Communication Device Driver Interface. The virtual communication interface is represented by our API, which provides all the services that are needed within our applications.

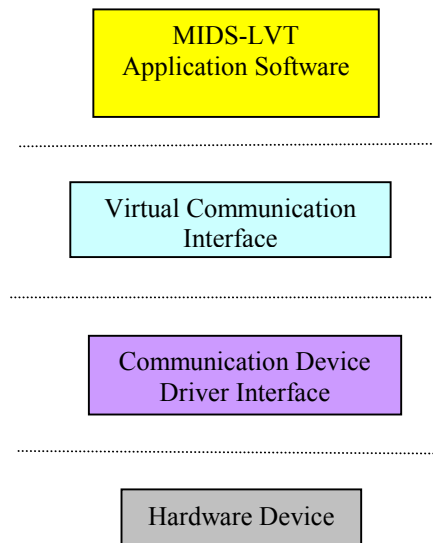


Figure 3.5. Communication layer interface

In figure 3.6, we show subclasses of the hardware wrapper, i.e., memory devices, communication devices, and others. Each device can be further extended to a specific type of hardware component with its additional unique interfaces, attributes, and methods as required.

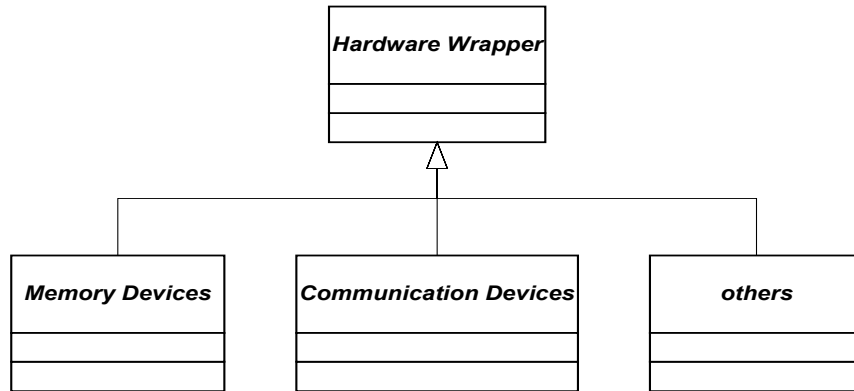


Figure 3.6. Hardware wrapper class

As shown in figure 3.7, the current MIDS-LVT inter-processor communications are conducted through shared memory, i.e., Core/MSG, Core/Voice, and Core/TIO. The shared memory consists of several data structures for the CSCIs to interact and exchange information. Other communication mechanisms include the MIL-STD-1553 (TIO/Host), the Ethernet (TIO/Host, Core/TE, and TIO/TE) and the RS-422 (MSG/RF subassembly).

We are extending the memory devices subclass to many specific memory types such as shared memory, Direct Memory Access (DMA), and reflective memory as shown in figure 3.8. The memory device subclasses inherit common methods from the memory devices, i.e., basic read, write, and initialization methods.

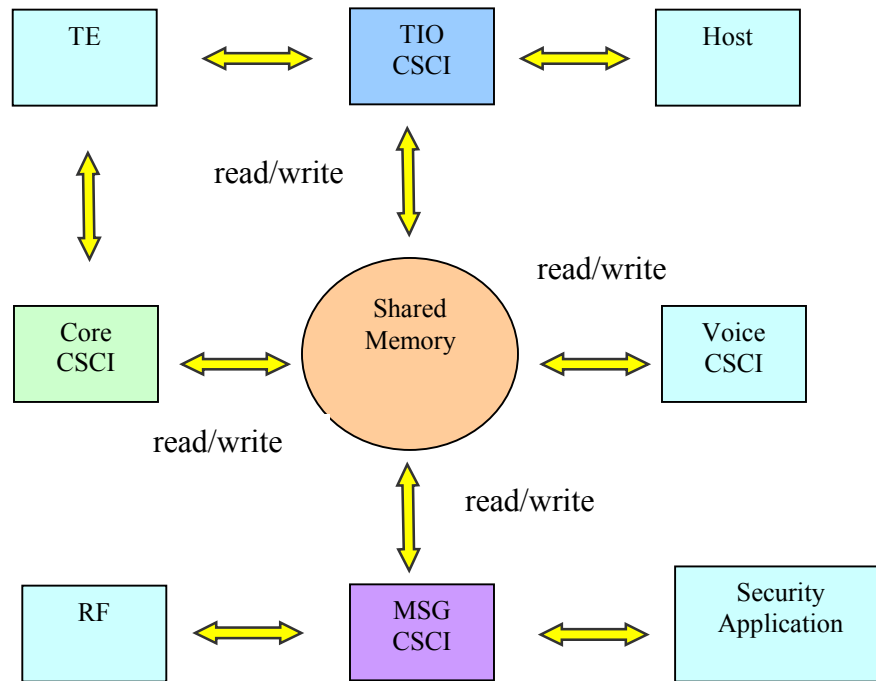


Figure 3.7. MIDS-LVT communication structure

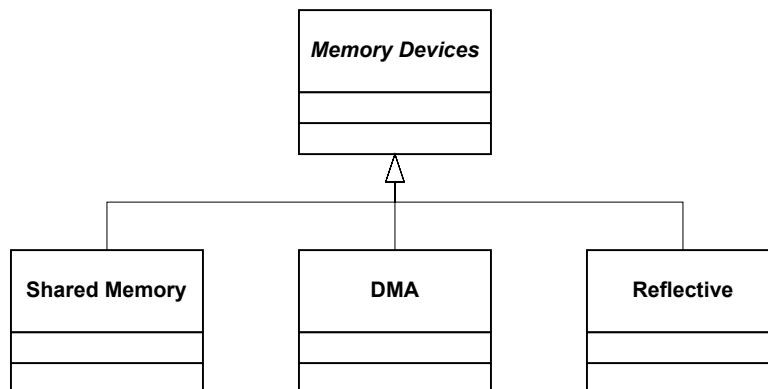


Figure 3.8. Memory devices class

B. INTEROPERABILITY MODEL

As complex real-time distributed systems, each CSCI cannot be independently developed and delivered as a plug-and-play software subsystem without extensive consideration of software interoperability.

In figure 3.9, we propose an interoperability model that is specific for the MIDS-LVT. This model consists of an interface specification, a protocol specification and a temporal specification. Based on our assessment of the current interoperability techniques, the MIDS-LVT CSCIs' interface specifications must be compatible at the level of signature and behavior, communication protocols, and temporal properties for MIDS-LVT CSCIs to be able to interact successfully.

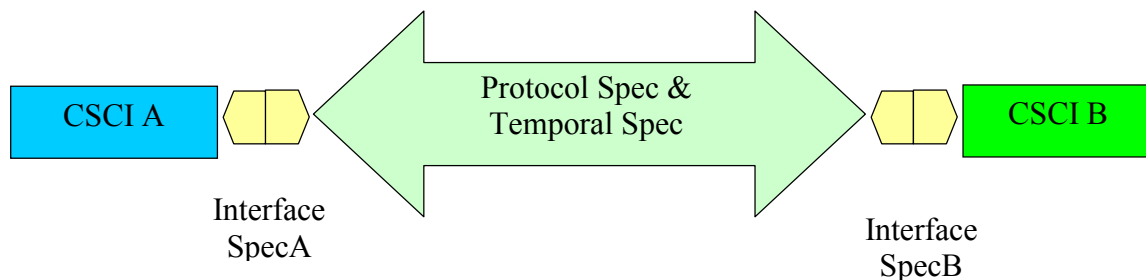


Figure 3.9. MIDS-LVT interoperability model

1. Interface Specification

Generally, an interface is a description of a set of possible operations that a client CSCI may request services through that interface. In object-oriented programming, an interface is a defined signature of methods and properties that can be implemented by a particular class. As a developer, if you were to implement the interface on your particular class you would need to provide all the methods and properties as defined by the signature. Failure to meet the interface specification would result in compilation errors.

In our model, each CSCI should have a well-defined interface to the other CSCIs in the system. Each interface specifies the form of all the interactions and the information

flow across the CSCI boundaries but does not specify how the CSCI is implemented internally. Each CSCI can then be redesigned independently without affecting the others.

The interface specification is represented by a set of APIs, which act as interfaces to facilitate CSCI interaction and cooperation in a distributed heterogeneous environment. The API provides a simple programming interface, which shields the software designer from the detail complex implementation of proprietary device drivers and RTOS facilities. As a result, changes and unsupported proprietary software are controlled to minimize the impact of the application software.

The CSCI defines the signature for each API method – the return type and the parameter types – supported by the CSCI. The behavior of the CSCI includes the role it plays and the pre and post-condition of each API method. For systems that use shared memory as the inter-processor communication mechanism, an interface specification is defined by a set of generic API that separate the specific detailed implementations that allow the CSCIs to interact with the other CSCIs through the shared memory. This allows us to define a clear separation between the behavior and the interaction of the CSCIs.

2. Protocol Specification

In general, protocol is a description of a set of mutually agreed upon conventions and procedures that govern what data to exchange and how to exchange. The interaction among the CSCIs can be accomplished by using communication protocols. Figure 3.10 shows a simple example of the MIDS-LVT interaction between the Core and the TIO CSCI. Record is viewed as a protocol, not a single message send from the TIO to the Core CSCI. The record protocol may consist of multiple messages.

The CSCIs interaction patterns can be modeled as client/server, peer/ peer, or multiple roles as shown in figure 3.11. In client/server model, one CSCI is always sending messages, and the second CSCI is always receiving messages. However, in peer to peer, the CSCI may act as a server at certain times and then act as a client at other times.

In our interoperability model, the communication protocol is defined as the sequence of messages involved in the interaction and cooperation of the two CSCIs. If the communication protocol between two CSCIs is not compatible, the CSCIs cannot interoperate. The use of the protocol provides a safe and verifiable information exchange between the CSCIs. It can be viewed as a strict constraint mechanism that controls the legal ordering of message and how and which messages can interact among the CSCIs. The protocol specification can be modeled using a finite state machine.

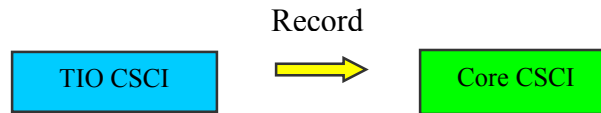


Figure 3.10. MIDS-LVT record protocol

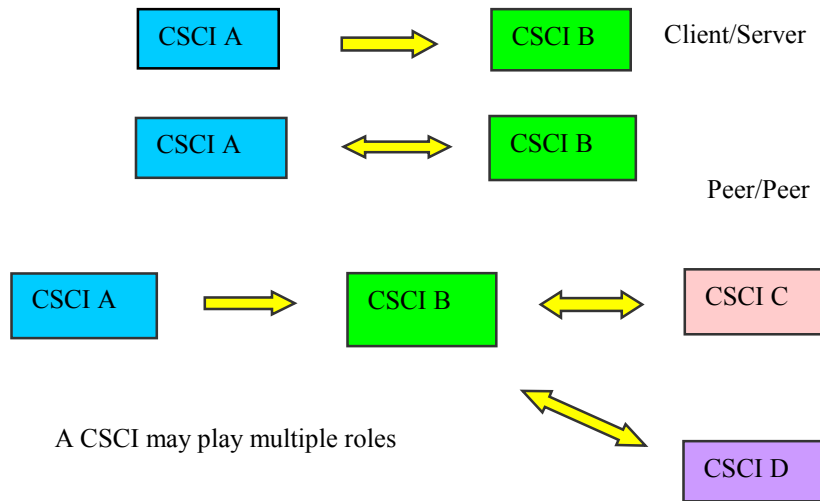


Figure 3.11. CSCI interaction patterns

3. Temporal Specification

Temporal properties are domain specific requirements. As the correctness of a real-time system depends not only on correct functions but also on correct timing, the temporal constraint must be presented in our interoperability model. When we talk about the temporal properties in our model, we are specifically interested in the ability of the

system to schedule the functions that provide and consume the data for interaction between each set of two CSCIs. For instance, when CSCI A requests processing data from CSCI B, the temporal requirement may require the output data from CSCI B to be made available to CSCI A within 100 microseconds after the request is submitted. In order to match the temporal properties, we must guarantee the availability of the data every time.

C. LOW LEVEL PROTOCOL SPECIFICATION

This section provides a detail explanation of the protocol currently used for communication in the legacy system. This protocol will be encapsulated in the APIs in Section E of this chapter. The APIs will decouple the application from the protocol and enable other protocols such as real-time CORBA to be substitute in the future without the need for additional changes to the application.

The data transfer between the Core CSCI and the TIO CSCI will be via shared memory. The Data Processor's VME shared memory contains two shared memory regions for the TIO CSCI to the Core CSCI messages. The Tailored Processor's VME shared memory contains a single shared memory region for the Core CSCI to TIO CSCI messages. Each CSCI will write messages to the other's shared memory via the VME bus. Each CSCI will read messages from the other CSCI into its own local memory. Each shared memory will occupy a contiguous region of physical memory.

The shared memory region consists of a Handshake Word, Pointer Words, and Data Transfer Blocks (DTBs) as shown in figure 3.12. A handshake word will be used to coordinate the transfer. The pointers are the offsets from the starting address of the shared memory region to the each DTB or message. A maximum of seven pointers is supported by the legacy systems. The DTBs are the actual message buffers.

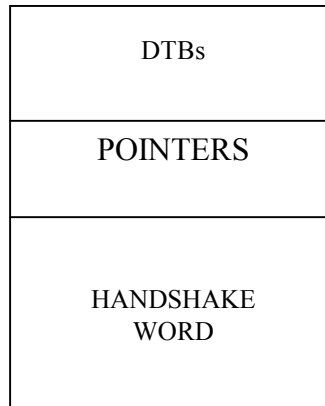
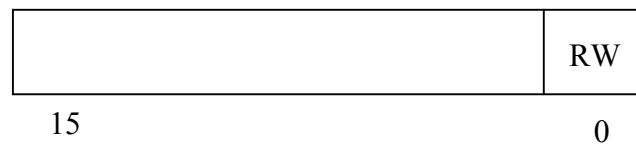


Figure 3.12. Shared memory region architecture

1. Handshake Word

The first region is the Handshake Word. The format of the Handshake Word is as follows:



Field READ/WRITE (RW)

Type: Coded

Value: 0 = Receiver Finished Reading

1 = Sender Finished Writing

Before writing to a shared memory region, the sender must make sure the READ/WRITE bit is set to Receiver Finished Reading. When the sender finishes writing

to a shared memory region, the sender must set the READ/WRITE bit to Sender Finished Writing.

Conversely, before reading from the shared memory region, the receiver must make sure the READ/WRITE bit is set to Sender Finished Writing. When the receiver finishes reading from a shared memory region, the receiver must set the READ/WRITE bit to Receiver Finished Reading. This handshake must be performed every slot even if the sender has no DTB to write to the shared memory region and receiver has no valid DTB to read from the shared memory region.

2. Pointers

The second section of the shared memory region contains pointers to the DTBs in the shared memory region. These pointers are word offsets from the absolute VME starting address of the shared memory region. Each pointer is 16 bits. The presence of a non-zero pointer indicates that a valid DTB is present at that location. The sender of the DTBs writes pointers after the DTBs have been written to the shared memory region. This enables the receiver to know the starting location of each DTB in the shared memory region. The receiver of the DTBs zeros pointers after the DTBs have been read from the shared memory region.

3. Data Transfer Blocks

The last section of each shared memory region contains the DTBs. DTBs are contained in the shared memory region in consecutive memory locations with no gaps between DTBs. A words count is provided in the header of each DTB to specify the length of the DTB. Message identification is also specified in the header of each DTB.

D. TEMPORAL SPECIFICATION FOR THE PROTOCOL

The temporal specification for the protocol for data transfer between the Core CSCI and TIO CSCIs can be divided into two phases as shown in figure 3.13:

Phase 1: Core I/O

- After receiving and processing the End Of Slot (EOS), the Core reads DTBs from the TIO-to-Core shared memory regions and writes DTBs to the Core-to-TIO shared memory regions

Phase 2: TIO I/O

- After receiving and processing the Data Transfer Interrupt (DTI), the TIO reads DTBs from the Core-to-TIO shared memory region and writes DTBs to the TIO-to-Core shared memory regions.

Phase 1 and phase 2 must not overlap in time. This is accomplished as follows. At the beginning of each time slot (period), the Core CSCI performs its I/O processing (with the TIO CSCI) and sends a Data Transfer Interrupt (DTI) to the TIO CSCI between 0 millisecond and 3.8 milliseconds maximum after the End-of-Slot (EOS). This is a signal to the TIO CSCI that it can perform its I/O processing (with the Core CSCI), which it must complete by the EOS.

Each CSCI will read the appropriate Handshake Word prior to reading from the appropriate shared memory region to ensure that the other CSCI has updated the region. Similarly, each CSCI will read the appropriate Handshake Word prior to writing to the appropriate shared memory region to ensure that the other CSCI has cleared the region. If the other CSCI has failed to perform its I/O processing, the CSCI must log the failure. Each CSCI will update the appropriate Handshake Word after the completion of the read or write.

If the interrupt from Core CSCI is later than 3.8 millisecond, the impact to other CSCIs is depending on the loading of the system at that time slot. In a normal load condition, this interrupt is generated around 3.5 millisecond. Beyond 3.8 millisecond, a

strong possibility that the system will start dropping messages. When this happen the terminal performance will start to degrade.

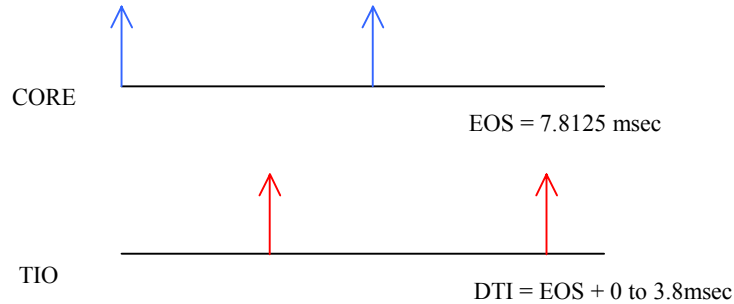


Figure 3.13. Timing diagrams of Core and TIO CSCI

E. INTERFACE SPECIFICATION

Generally, an API defines what data structures and facilities are available for use by the application program without defining how the structure and facilities are implemented. In the component-based software technology, APIs are critical for any vendor implementing a complex system who needs to cleanly partition work effort, migrating code from one platform to another, and abstract away interfaces so hardware changes can be made easily. Additionally, the use of APIs – especially standardized APIs – makes possible a whole set of vendors creating different elements of the system, interfacing to multiple hardware vendors, other software vendors and customer-developed elements.

In the MIDS-LVT system, APIs are the standardized interfaces that present inter-processor communication functionality via shared memory to the rest of the software architecture. For this reason, APIs are specified in a language-independence fashion. We define six basic interface connection services for shared memory. These interface services inherit and extend from the Connection base class for the application to access the shared memory as shown in figure 3.14. The Connection interface is abstract class that consists of three basic services: isDeviceOK, Read, and Write. These services are pure virtual functions in C++ language. Most of the shared memory APIs are intuitive.

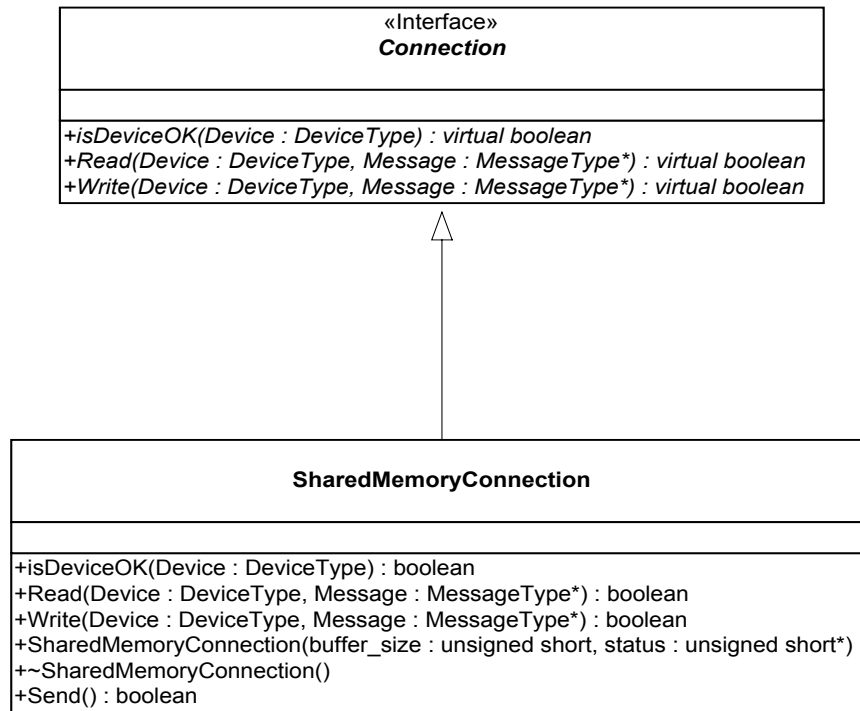


Figure 3.14. API services for shared memory

F. API SERVICES FOR THE SHARED MEMORY CONNECTION

1. isDeviceOK

The purpose of the API is to perform a built-in-test and initialize the buffer for the CSCIs' inter-processor communication. This function fails if the buffer is not empty.

Syntax:

isDeviceOK(Device : DeviceType) : boolean

Parameters:

Device – This parameter is a device type (e.g., shared memory).

Response:

true - if shared memory is successfully initialized and passes the write/read built-in-test.

false - if the buffer is not successfully initialized or fails the write/read built-in-test.

State:

This command is valid in all states.

New State:

This command causes a state change.

Originator:

The service user.

2. Read

The purpose of the API is to read the next available message from the buffer. Before starting any elaboration, the API will check if the sender CSCI has finished writing to the receiver CSCI's buffer. If it is not the case, the API will return an error. After the API has completed reading all the messages, it will set the handshake word to Receiver Finished Reading, which informs the sender CSCI that the receiver CSCI has finished reading the messages (unlock the shared memory).

Syntax:

Read(Device : DeviceType, Message : MessageType) : boolean*

Parameters:

Device – This parameter is a device type (e.g., shared memory).

Message – This parameter is a data buffer allocated by the caller. The contents of the shared memory buffer are copied to this buffer. The procedure uses the word count fields of the shared memory data buffer to determine the numbers of words to copy.

Response:

true - if the receiver CSCI read the message from the sender CSCI successfully.

false - when a generic error occurs.

State:

This command is valid in all states.

New State:

This command causes a state change.

Originator:

The service user.

3. Write

The purpose of the API is to write a message into the specific receiver buffer. In case there is not enough room to store the message, the API will not store anything. Before starting any elaboration, the API will check if the CSCI can write in the specified receiver CSCI's buffer. If it is not the case, the API will return false, indicating failure to write.

Syntax:

Write(Device : DeviceType, Message : MessageType) : boolean*

Parameters:

Device – This parameter is a device type (e.g., shared memory).

Message – This parameter is a data buffer allocated by the caller. The contents of the buffer are copied to the shared memory buffer. The procedure uses the word count fields of this buffer to determine the numbers of words to copy.

Response:

true - when the message has been stored in the buffer successfully.

false - when a generic error occurs.

State:

This command is valid in all states.

New State:

This command causes a state change.

Originator:

The service user.

4. Send

After the Write API has completed writing all the messages to the shared memory, this API will set the handshake word to *Sender Finished Writing* which informs the receiver CSCI that the sender CSCI has finished writing the messages (unlock the shared memory). It will also trigger the Data Transfer Interrupt to inform the receiver CSCI. This API is needed to synchronize the messages to a specific time slot. For example, if the shared memory contains five messages, these messages must send and read at the same time slot. The messages may contain navigation or tracks data which require precise correlation by the host platforms with other sensors.

Syntax:

Send() : boolean

Parameters:

None.

Response:

true - if the indication is generated successfully.

false - when a generic error occurs.

State:

This command is valid in all states.

New State:

This command causes a state change.

Originator:

The service user.

5. SharedMemoryConnection

This API is the constructor for the *ShareMemoryConnection*. The purpose of the API is to allocate and check memory for the shared memory object when it is first created. This API prevents the user from creating two or more shared memory objects with the same address space. The status word parameter is passed as a pointer to the API. In case there is not enough room to create the shared memory object, the API will return an error in the status word.

Syntax:

SharedMemoryConnection(buffer_size : unsigned short, status : unsigned short)*

Parameters:

buffer_size – The parameter is the buffer size in the shared memory object (16-bit word).

status – The parameter is a status word passed by pointer to the API. The return value indicates *SUCCESS* (1) or *FAILURE* (0).

Response:

SUCCESS - if the shared memory object is successfully allocated and checked.

FAILURE - if the shared memory object cannot be allocated and checked.

State:

This command is valid in all states.

New State:

This command causes a state change.

Originator:

The service user.

6. ~SharedMemoryConnection

This API is the destructor for the *ShareMemoryConnection*. The purpose of the API is to release the memory used the shared memory object when the connection is no longer needed, such as when application terminates.

Syntax:

~SharedMemoryConnection()

Parameters:

None

Response:

None.

State:

This command is valid in all states.

New State:

This command causes a state change.

Originator:

The service user.

G. SUMMARY

This chapter presents the MIDS-LVT interoperability model for CSCI inter-processor communication in a heterogeneous distributed environment. Our model consists of the API, the protocol, and the temporal specification that are needed for CSCI interoperation. The API allows us to separate the CSCI's internal activity from its external relationships. The protocol provides a strict constraint mechanism and policy to control the legal ordering of the sequence of messages involved in the interaction of the MIDS-LVT CSCIs. The temporal specification provides the timing requirements and constraints for the interactions of the MIDS-LVT CSCIs.

We also present the top-level architecture framework for the MIDS-LVT and the six APIs for the CSCI inter-processor communication via the virtual communication interface to the shared memory. The API functions are grouped into building blocks to define the inter-processor communication services, which foster software reuse and commonality among the CSCIs.

APIs are critical for any vendor implementing a complex system who needs to cleanly partition work effort, migrating code from one platform to another, and abstract away interfaces so hardware changes can be made easily. Additionally, the use of APIs – especially standardized APIs – makes possible a whole set of vendors creating different elements of the system, interfacing to multiple hardware vendors, other software vendors and customer-developed elements.

In Chapter IV, we will present the design and implementation of our API.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. DESIGN AND IMPLEMENTATION

This session describes our APIs architecture and its implementation details.

A. API ARCHITECTURE DESIGN

The architecture design pattern represented in figure 4.1 is the approach for the design and implementation of our API for the MIDS-LVT inter-processor communication. We use the facade pattern, which provides a unified interface to a set of objects in the hardware devices. This pattern defines our API, which is a higher-level interface that makes the hardware device easier to use, i.e., it abstracts out the gory details. We use aggregation for the hardware devices whose parts are a set of APIs in the façade class. We are avoiding client direct access to the hardware devices. All client service requests have to pass through the façade class.

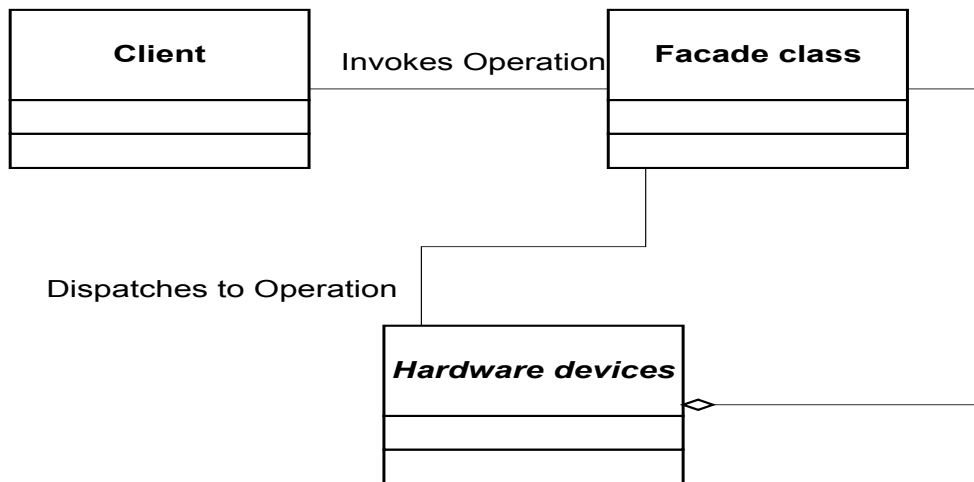


Figure 4.1. Architecture design pattern

In figure 4.2, the collaboration diagram shows the interaction between the client, the façade controller, the hardware devices and the specific type of devices. Where the class diagram defines a static relationship structure between the classes, the collaboration diagram defines a communication structure between the objects of those classes.

Whenever the façade-controller receives a dispatchable service request, it forwards the request to the appropriate message dispatcher (hardware devices class). The dispatcher gathers any necessary information and then dispatches the request to the appropriate subclass (shared memory class).

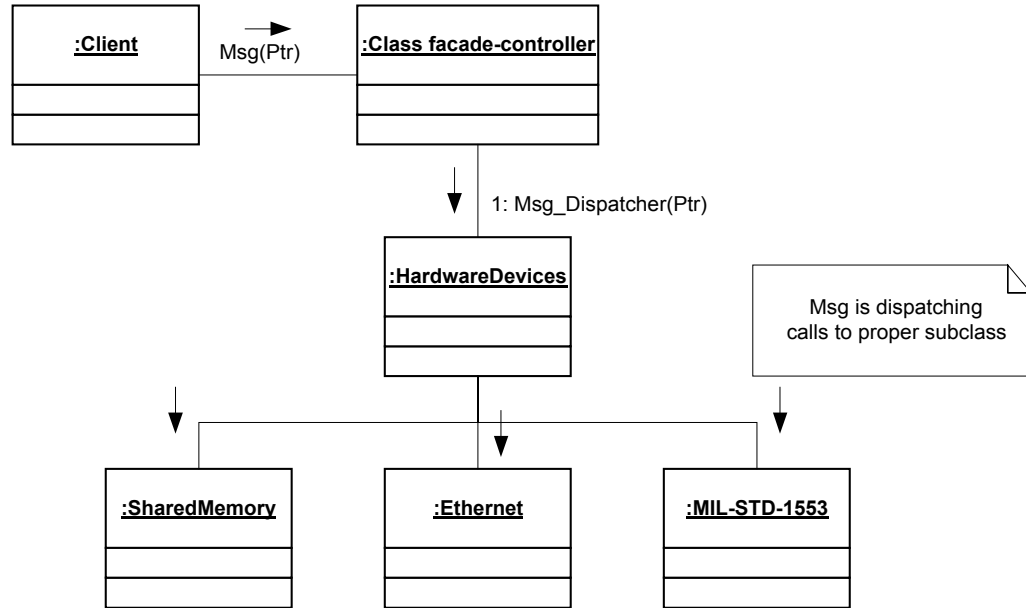


Figure 4.2. Collaboration diagram of API

B. API PROTOCOL DESIGN

The class diagram presents a static view of our API. To understand the behavior of the API for the MIDS-LVT we created new diagrams showing the aspect of our design. The statechart, collaboration, and message sequence diagrams describe the dynamic behavior of the API. We will use statecharts to shows the constructor, isDeviceOK, read, write, and send protocol.

1. Constructor (SharedMemoryConnection) Protocol

As shown in figure 4.3, a constructor protocol controls the shared memory allocation process. The constructor prevents the user from allocating the same shared memory region a second time after it had been allocated. *SharedMemoryConnection* is allowed to execute only once, from the initial state.

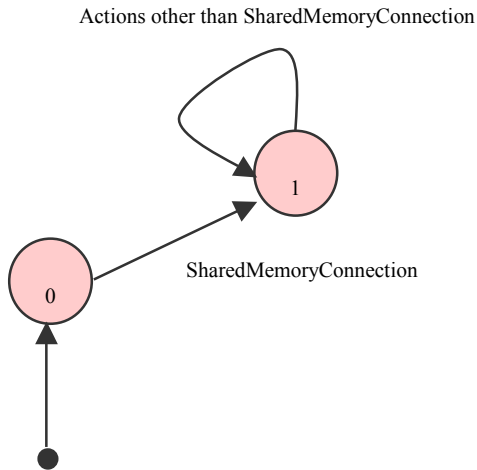


Figure 4.3. Constructor (SharedMemoryConnection) protocol statechart

2. isDeviceOK Protocol

As shown in figure 4.4, the *isDeviceOK* protocol performs the write/read built-in-test and initializes the buffer. The transition from state S0 to S1 indicates that an error condition has occurred. The memory device failed the write/read built-in-test. If this case happens, the application may need to allocate new memory that maps to different region of the physical address. In the case of a hardware device (MIL-STD-1553 and RS-422) fail, the system's performance may degrade.

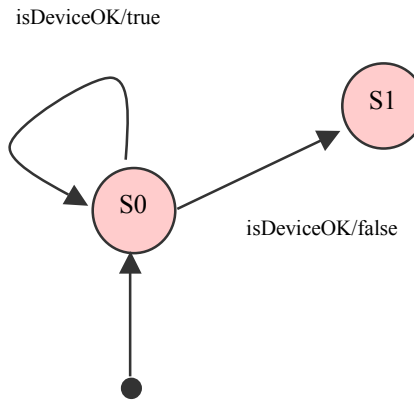


Figure 4.4. isDeviceOK protocol statechart

3. Write Protocol

A Write protocol controls how data is written to the shared memory as shown in figure 4.4. The transition from state S0 to S1 requires that we have enough room in the shared memory for the message. We also need to check the handshake word to make sure that the receiver has finished reading the previous data. If the response is fail, the return status will indicate an error condition. In this case, no more room is available in the shared memory object. If the response is success, we can transition into state S1 and back to S0. Before going back to state S0, we will write data to the shared memory buffer. The write operation enforces the protocol of checking to make sure only one processor can enter the critical section using the *isFinishedRead* function. This is the pre-condition. The post-condition, is the execution of the *setFinishedWrite* function, which is part of the Send protocol required to unlock the critical section of the shared memory for the other processor to access.

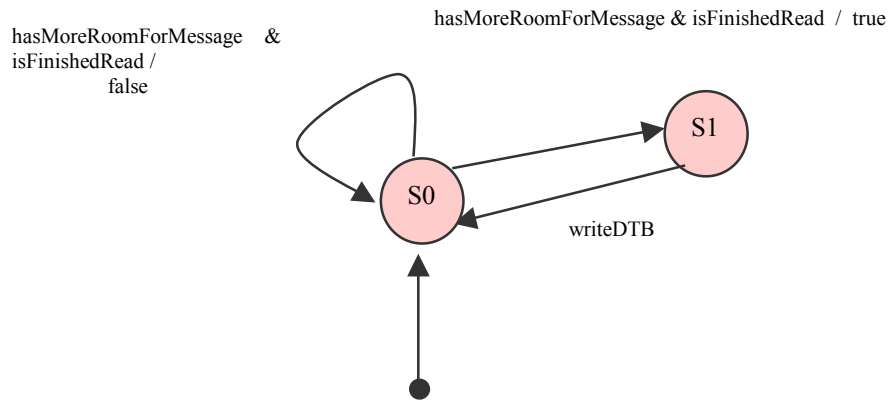


Figure 4.5. Write protocol statechart

4. Send Protocol

As shown in figure 4.6, a send protocol controls when the data buffers in the shared memory are send. The transition from state S0 to S1 indicates successful unlocking of the shared memory. If the response is fail, the return status will indicate an error condition. If the response is success, we can transition back to state S0. Before going back to state S0, we will trigger the Data Transfer Interrupt to inform the receiver CSCI that we are finished writing to the shared memory.

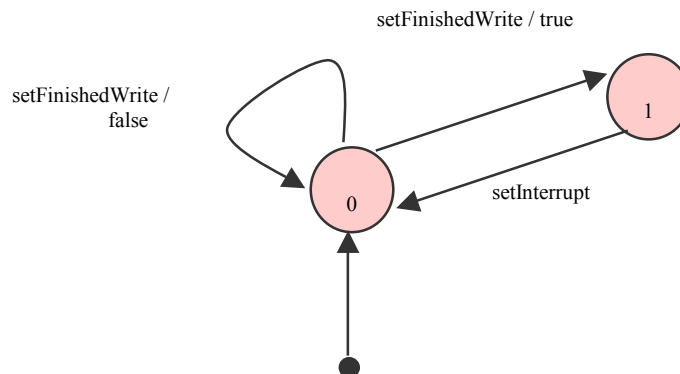


Figure 4.6 Send protocol statechart

5. Read Protocol

As shown in figure 4.7, a Read protocol controls how data is read from the shared memory. The transition from the initial state to state S0 requires a message “Event”. For the TIO CSCI, the event is a software interrupt (DTI) generated by the Core CSCI (*Write API*) when the last buffer is written into the shared memory. For the Core CSCI, the event is periodic hardware interrupt (EOS) generates every 7.8125 millisecond, which indicates the ending of a slot. The TIO CSCI must complete its *Write API* before the EOS is generated.

In state S0, we check to make sure that the sender had finished writing data to the shared memory. If the response is fail, an error condition will occur. In this case, we need to go back to state S0 and wait for a new event message. If the response is success, we will transition to state S1. In this state, we will need to check that there are no more messages available to be read from the shared memory. If the response is false, we go to state S2 and back to S0. If the response is true, we go to state S3. This could be an error condition or it could just mean that the sender wrote no data or no more messages to be read. Before going back to state S0, we will set the Receiver Finished Reading bit in the handshake word to informs the sender CSCI that the receiver CSCI has finished reading the messages (unlock the shared memory).

Similar with the write protocol, *IsFinishedWrite* is the pre-condition and *SetFinishedRead* is the post-condition required for the receiver not to enter the critical shared memory section while the sender is still accessing it. This is critical for our application because our data buffer is tied to a specific time slot.

If the response is false, this indicates that valid data is present at that location and we transition to state S2 and back to S0. We will execute as many read operations as required to get all the messages from the shared memory.

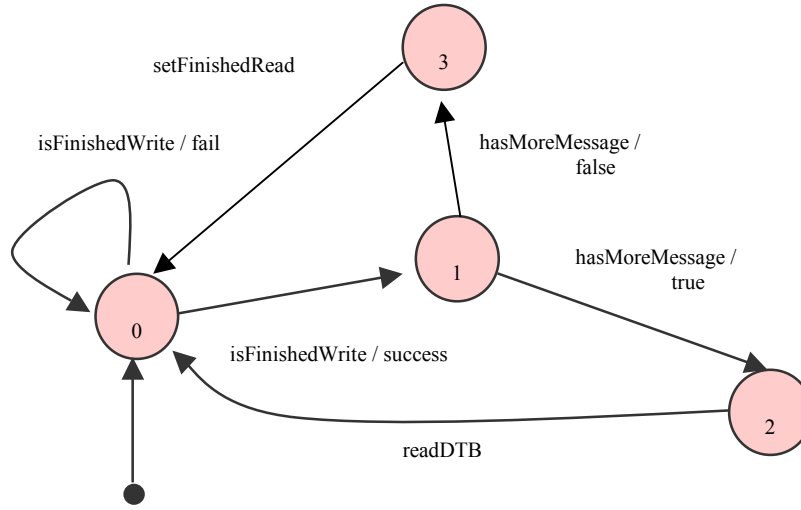


Figure 4.7. Read protocol statechart

C. IMPLEMENTATION

This section describes the concrete implementation details. The concrete implementation of the APIs was implemented under the Microsoft Visual C++ Version 6.0 programming language. This implementation conforms to the interface and protocol specifications of our interoperability model. Since no hardware or operating system related facilities are used in our APIs' implementation, they are portable.

In figure 4.8, we present the UML class diagram of our APIs. The APIs' functions implemented in C++ for data transfer for shared memory in a multiprocessor environment are as follows:

1. Constructor

The *Constructor* method takes two parameters – *size* and *status*. The *Size* parameter is the size of the shared memory object. The *status* parameter is the status of the allocation of memory for the shared memory object. *Status* is an *unsigned short*,

which is passed by pointer to the constructor. The return value of the *status* parameter represents the success or failure of the constructor execution.

```
SharedMemoryConnection::SharedMemoryConnection(unsigned short buffer_size,  
unsigned short* status)
```

The constructor communicates with a static function in the *ShmManager* class to allocate physical memory to the shared memory object.

```
shm_addr = ShmManager::allocate( buffer_size, &allocate_status)
```

Allocate is a static method that take two parameters – *buffer_size* and *allocate_status*. The *Buffer_size* is the size of the shared memory object. The *Allocate_status* is a pass-by pointer and returns the status from the *allocate* method. The status value is *one* if it succeeded in completing the actions, otherwise it return a *zero* value. This static method returns the address of the shared memory object (*shm_addr*) after completing the allocate actions.

The constructor also call *isDeviceOK* after it successfully allocate memory region.

2. IsDeviceOK

The *isDeviceOK* method takes one parameter – *Device*. The *Device* parameter is an enumerate type that list of all the possible devices (shared memory, Ethernet, Mil_STD-1553, and RS-422). For our API, the *Device* parameter consists of a shared memory. This method returns a *true* value if it succeeded in completing the actions, otherwise it return a *false* value.

```
bool SharedMemoryConnection::isDeviceOK(DeviceType Device)
```

3. Read

The *Read* method takes two parameters – *Device* and *Message*. The *Device* parameter is an enumerate type that list of all the possible devices (shared memory, Ethernet, Mil_STD-1553, and RS-422). For our API, the *Device* parameter consists of a shared memory. The *Message* parameter is a message buffer type that contains data and

methods that needed to be transfer to and from the shared memory. This method returns a *true* value if it succeeded in completing the actions, otherwise it return a *false* value.

```
bool SharedMemoryConnection::Write( DeviceType Device, MessageType* Message)
```

4. Write

The *Write* method takes two parameters – *Device* and *Message*. The *Device* parameter is an enumerate type that list of all the possible devices (shared memory, Ethernet, Mil_STD-1553, and RS-422). For our API, the *Device* parameter consists of a shared memory. The *Message* parameter is a message buffer type that contains data and methods that needed to be transfer to and from the shared memory. This method returns a *true* value if it succeeded in completing the actions, otherwise it return a *false* value.

```
bool SharedMemoryConnection::Write( DeviceType Device, MessageType* Message)
```

5. Send

The *Send* method takes no parameter. This API sets the handshake word to *Sender Finished Writing* which informs the receiver CSCI that the sender CSCI has finished writing the messages (unlocks the shared memory). This method returns a *true* value if it succeeded in completing the actions, otherwise it return a *false* value.

```
bool SharedMemoryConnection::Send()
```

6. Destructor

The *Destructor* method takes no parameter. This API will release the memory used in the shared memory object when the application terminates

```
SharedMemoryConnection::~SharedMemoryConnection()
```

This implementation uses pointers to access the message objects in the shared memory. A set pointer is sent from the producer processor to the consumer processor. The pointers point to the buffers that are in the shared memory region accessible to both processors. The usage of pointers is straightforward and efficient. This is because the

address of a shared object in one processor is the same as that in the other processor. Thus, the programmers are not required to translate between local and global addresses of a shared object.

An ordinary access requires that the programmer must follow the protocol to ensure the correctness. The protocol provides synchronization and mutual exclusion guarantees that the consumer will obtain the most up-to-date data available at the time of the consuming. Specifically, our protocol guarantees that all message buffers are written and read as a group within the specific time slot. This is a robustness requirement of our system. The full source code is provided in Appendix A.

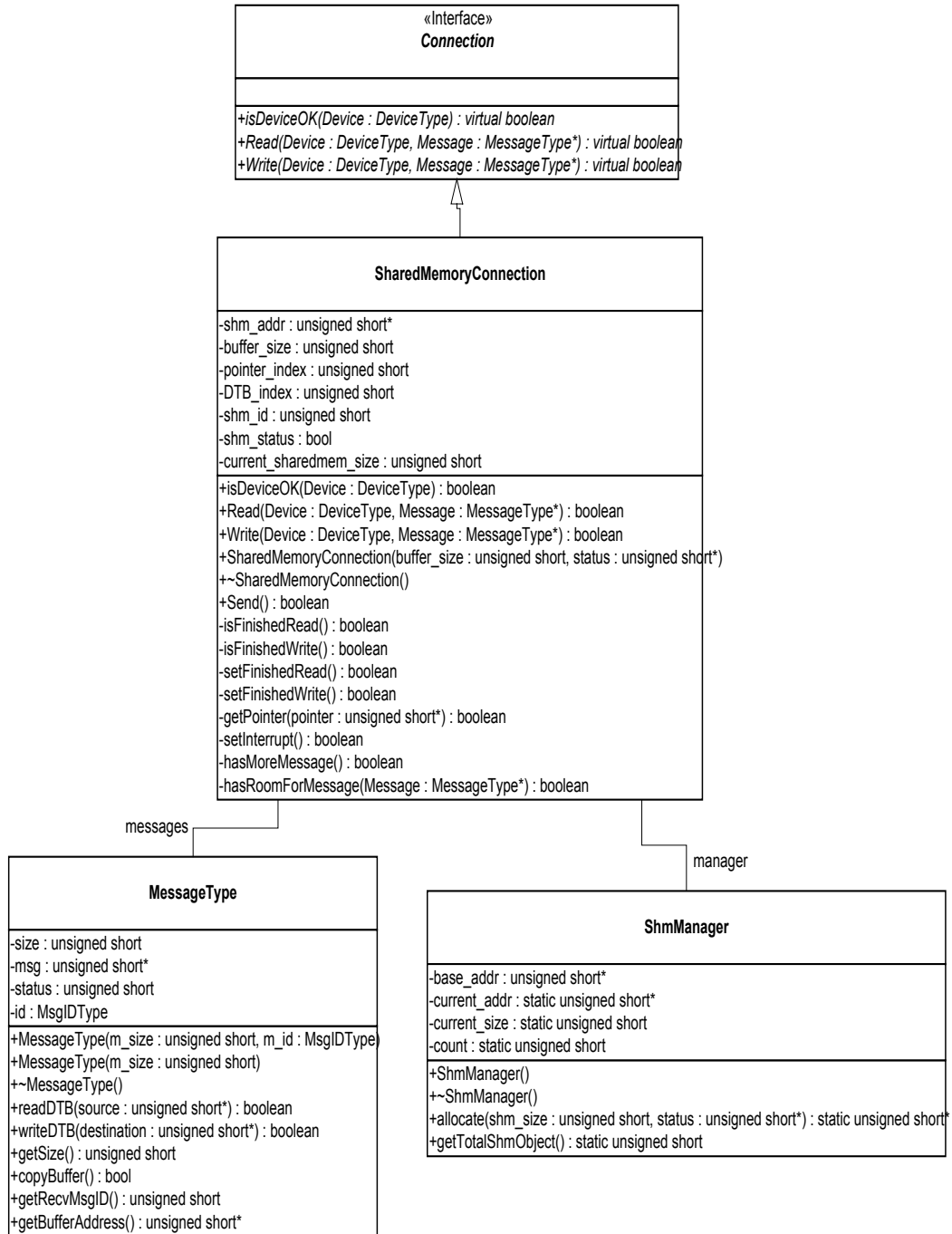


Figure 4.8. API UML diagram

THIS PAGE INTENTIONALLY LEFT BLANK

V. RESULTS

In this section, we present the results of an experiment conducted to show the performance of API services for inter-processor communication.

A. TEST ENVIRONMENT AND METHODOLOGY

We implemented and tested the first version of the API on our simulated embedded distributed environment for the Core and TIO CSCIs. Our simulated system consists of a MZ7140 VMEbus Single Board Computer, a BIU-153V, and a VBT-325 as shown in figure 5.1. The MZ7140 is a Single Board Computer, which consists of a MC68040 @ 25MHZ, four Mbytes of multiple-access DRAM, an on-board SCSI, and Ethernet interfaces. The BIU-153V is a Bus Interface Unit that provides a connection between a host and the MIL-STD-1553 bus. The BIU-153V has a high-speed controller in conjunction with Dual Port Random Access Memory, which was used as the shared memory for testing the API inter-processor communication software. The VBT-325 is a bus analyzer for VME that provides capabilities such as state analysis, timing analysis, and statistical analysis. Its application includes hardware and software debugging and testing, system tuning, and performance analysis.

The test software API was coded in C and compiled using the Microware Version 1.3 of the Kernigham and Ritchie (non ANSI/ISO-conformant) C compiler for the OS-9 real-time operating system. Since no hardware and operating system related facilities are used in the test program this program is portable to other RTOSs.

Timings were obtained using a software-readable hardware counter in the VBT-325, which measured elapsed time. The elapsed time is calculated based on trigger and store conditions. The VBT-325 uses the event patterns and sequencer as the main control elements to define the trigger and store conditions. The event patterns define a trigger, store or count conditions and the sequencer defines a complex trigger condition, store qualifier, etc.

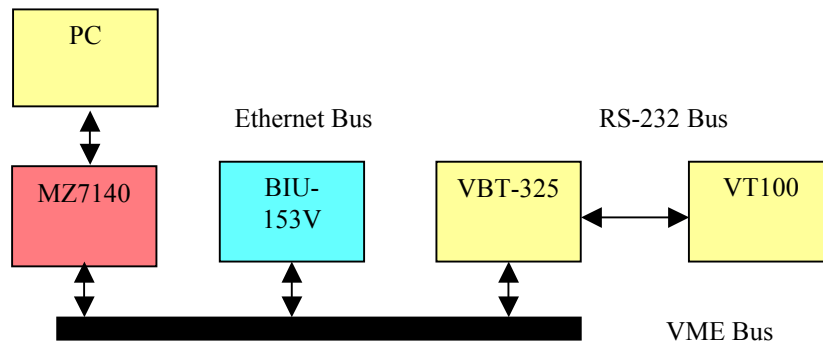


Figure 5.1. Test environment

B. PERFORMANCE EVALUATION

The main focus of the performance tests is on how much overhead or latency does the API really incur compared with the legacy approaches as shown in figure 5.2. The latency or overhead may be loosely defined as the time from when an API calls occurs until it is serviced. In a multi-processes environment this time can vary for a number of reasons. First, the CPU will always finish the current instruction before servicing the API call, and some instructions can take longer than others. The CPU may be executing a sequence of instructions protected by a high priority thread or the CPU may be executing an interrupt service routine, which often has interrupts disabled. The actual timings in a normal operation may vary considerably depend on the state of the system and its hardware.

The overhead costs of the API calls are important in order to obtain a characterization of the implementation on various real-time operating systems. Since it is difficult to measure the overhead for round-trip communication in our simulated embedded distributed environment without the synchronized clock (DTI and EOS), our measurements were all made on one-way communications. In particular, our overhead measurements were done with respect to one-way write and read API interfaces and protocols.

To measure the overhead of legacy approach, we recorded a direct write and read to the shared memory without the use of the API. The data is present in table 5.1. The average measured time for two bytes of data during read is 1.21 microsecond and during write is 1.19 microsecond. As expected, there is no overhead with the direct access to the shared memory.

To compare the performance of the API with the legacy approaches, each measurement includes the time to execute the following steps:

1. Writing steps

The writing steps for the measurement of one-way latency is as follows:

- The Writer calls *IsFinishedRead* to acquire the lock.
- If it is true, the Writer then starts sending data to the shared memory.
- When completed writing, the Writer then performs *SetFinishedWrite* to release the lock.

2. Reading steps

The reading steps for the measurement one-way latency is as follows:

- The Reader calls *IsFinishedWrite* to acquire the lock.
- If it is true, the Reader then starts reading data from the shared memory.
- When completed reading, the Reader then performs *SetFinishedRead* to release the lock.

Table 5.2 shows the average measured times to acquire and release a lock. To acquire the lock the average measured time was 4.19 microseconds and to release the

lock the average measured time was 5.09 microseconds. The operations consisted of reading, checking, or writing two bytes of data into the shared memory.

In table 5.3, we show the average time measured for both writing and read steps. It indicates that the average one-way communication time is almost proportional to the message size. As result, the average overhead of using the API is almost constant. For reading steps, the average overhead is 8.9 microseconds and for writing steps, the average overhead time is 9.1 microseconds.

Comparing table 5.1 with table 5.3, the measured timing results indicate that the use of API incurred very little extra overhead. From Table 5.1, we calculated the average measured time of transferring 512 words to or from the shared memory without using the API and it took about 614 microseconds. From Table 5.3, we also calculated the average measured time for transferring 512 words to or from the shared memory using the API and it took about 626 microseconds. The different is 14 microseconds. Therefore, by using the API we incurred only about two percent of overhead.

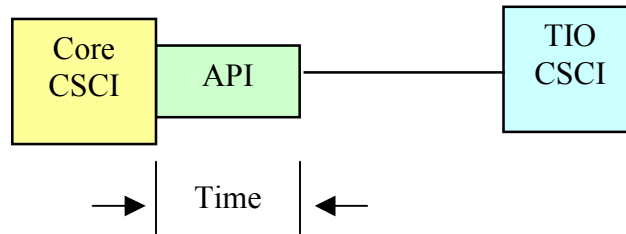


Figure 5.2. Timing overhead

Operation	Message size (bytes)	Elapse Time (us)
Read	2	1.21
	1024	620
Write	2	1.19
	1024	609

Table 5.1 Read and write without API

Operation (API)	Message size (bytes)	Elapse Time (us)
Acquire	2	4.19
Release	2	5.09

Table 5.2 Time to acquire and release a lock using API

Operation (API)	Message size (bytes)	Elapse Time (us)	Overhead (us)
Read	2	12.5	8.9
	1024	631	8.9
Write	2	12.7	9.1
	1024	621	9.1

Table 5.3 Overhead benchmarks for reading and writing steps (one-way communication)

THIS PAGE INTENTIONALLY LEFT BLANK

VI. DISCUSSION

A. PROGRAMMING LANGUAGE

We have chosen the C programming language instead of other languages to implement and test our API simply because it was the only available distributed embedded development environment for us to conduct our experiment. This implementation could have easily been done in other object-oriented languages (OOL) such as C++, Java, and Ada under different development and RTOS environments.

We also realized that the chosen specific programming language, hardware, and RTOS environment for implementation will affect the overhead costs for the API. For example, dynamic polymorphism occurs when the binding of the executable code to the operator invocation is done as the program executes. Depending on the chosen OOL, hardware, and RTOS environment, the API performance may vary significantly.

B. PROGRAMMING NOTES

This section provides general notes and examples to assist the user in the use of the *SharedMemoryConnection* API written in C++ language.

In the MIDS-LVT, *SharedMemoryConnection* APIs are the standardized interfaces that present inter-processor communication functionality via shared memory to the rest of the software architecture. The following examples use the APIs to illustrate how an application should allocate, check, read, write, and send. Please refer to chapter III, section C and chapter IV, section C of this thesis for detailed explanations of the low-level protocol specification and the APIs and their parameters.

- 1. Allocating the shared memory:** The shared memory must be requested and allocated before access to use the shared memory object is permitted. The constructor of the shared memory object is responsible for allocating and checking memory from the shared memory manager. If the allocation fails, all other calls to the shared memory

object will fail as well. In the following example, the shared memory object is been allocated and checked.

Declare a pointer (myshmgr) to the shared memory manager (ShmManager) and request memory from the shared memory manager. Declare a pointer (myshm) to the shared memory connection (SharedMemoryConnection) and request memory for the shared memory connection. The return status from the constructor indicates success (1) or failure (0).

```
ShmManager* myshmgr;  
myshmgr = new ShmManager();  
  
SharedMemoryConnection* myshm;  
myshm = new SharedMemoryConnection(shm_max_size, &status);
```

2. Checking the shared memory: The *isDeviceOK* API performed the write/read built-in-test to the allocated shared memory region. If the check fails, we may have a bad memory region. The user should try to re-allocate a new region, otherwise all data transfers may be corrupted.

The *isDeviceCheck* is a member function of *SharedMemoryConnection*. The return boolean status indicates success (true) or failure (false).

```
// check device using write/read built-in-test  
if (isDeviceOK( Sharedmem ))  
{  
    ...  
}
```

3. Writing message buffer: The *Write* API writes each message buffer to the shared memory region. In case there is not enough room to store the message, the API will not store anything and will return a fail status. In this case, we may have a corrupted message buffer. For example, the word count the message buffers might be wrong. The user may want to drop this particular set of messages and wait for the next time slot.

The *Write* is a member function of *SharedMemoryConnection*. The return boolean status indicates success (true) or failure (false).

```
// write message to the shared memory
if (!(myshrmem->Write( Sharedmem, dtb1_out)))
    cout << " Error, write fail \n";
```

4. Sending message buffer: After the *Write* API has completed writing all the messages to the shared memory, the *Send* API informs the receiver CSCI that the sender CSCI has finished writing the messages. No data will be transfer until this API executes. This is an important concept for the MIDS-LVT. We must synchronize a set of messages to a specific time slot.

The *Send* is a member function of *SharedMemoryConnection*. The return boolean *status* indicates success (true) or failure (false).

```
//send the message buffers at once
if (!(myshrmem->Send()))
    cout << " Error, send fail \n";
```

5. Reading message buffer: The *Read* API will read the next available message from the shared memory region. After this API has completed reading all the messages, it informs the sender CSCI that the receiver CSCI has finished reading the messages.

The *Read* is a member function of *SharedMemoryConnection*. The return boolean status indicates more message (true) or no more message (false).

```
// read while message is available in the current time slot
while (myshrmem->Read( Sharedmem, dtb_in) )           // return true or false
{
    switch (dtb_in->getRecvMsgID())                     // get the message ID
    {
        case DTB1:                                     // DTB1
            // copy form DTB to local buffer DTB1
            dtb_in->copyBuffer(dtb1_in->getBufferAddress(), size_dtb1);
```

C. EXTENDING INTEROPERABILITY MODEL

Our model can be extended to other CSCIs in the MIDS-LVT. We will explore this from an abstract viewpoint for the Core CSCI/MSG CSCI, the MSG CSCI/RF subassembly, the TIO CSCI/Host, the Core CSCI/Voice CSCI, the Core CSCI/Terminal Exerciser, and the TIO CSCI/Terminal Exerciser interfaces. Extending our model is possible due to our interface inheritance which separates each application from its internal detail implementation.

1. Core CSCI and MSG CSCI

The Core and MSG CSCIs communicate using the shared memory that resides in the DP SRU. The physical base address is the same as for the Core and TIO CSCI. Using the same API, we can easily implement the new requirement for the Core and MSG communication.

2. Core CSCI and Voice CSCI

Communication between the Core and Voice CSCI is accomplished through using the dual-port shared memory that resides in the Voice SRU. Our API and methods will remain the same. The physical base address of the memory in the Voice SRU must be coded in the program for this shared memory object.

3. MSG CSCI and RF Subassembly

The MSG CSCI communicates with the RF subassembly via an RS-422 bus. Our API will remain the same as shown in figure 6.1. The specific detailed implementation will change according with the MSG/RF subassembly device, protocol, and its buffer architecture. The *Device* type is *RS-422*.

RS422Connection
+isDeviceOK(Device : DeviceType) : boolean +Read(Device : DeviceType, Message : MessageType*) : boolean +Write(Device : DeviceType, Message : MessageType*) : boolean +RS422Connection(buffer_size : unsigned short, status : unsigned short*) +~RS422Connection() +Send() : boolean

Figure 6.1. MSG/RF subassembly API

4. TIO CSCI and Host

The TIO CSCI sends and receives data from various hosts using the MIL-STD-1553 and Ethernet protocols. Our API will remain the same as shown in figure 6.2. The specific detailed implementation will change according with the TIO/Host device, protocol, and its buffer architecture. The *Device* type is *MIL-STD-1553* or *Ethernet* depending on the host platform.

MIL-STD-1553Connection
+isDeviceOK(Device : DeviceType) : boolean +Read(Device : DeviceType, Message : MessageType*) : boolean +Write(Device : DeviceType, Message : MessageType*) : boolean +MIL-STD-1553Connection(buffer_size : unsigned short, status : unsigned short*) +~MIL-STD-1553Connection() +Send() : boolean

EthernetConnection
+isDeviceOK(Device : DeviceType) : boolean +Read(Device : DeviceType, Message : MessageType*) : boolean +Write(Device : DeviceType, Message : MessageType*) : boolean +EthernetConnection(buffer_size : unsigned short, status : unsigned short*) +~EthernetConnection() +Send() : boolean

Figure 6.2. MID-STD-1553 and Ethernet API

5. Core CSCI and Terminal Exerciser

The Terminal Exerciser is field test equipment that is required for the reprogramming and recording of internal MIDS data. The communication between the Core CSCI and the Terminal Exerciser can be accomplished via the Data Processor Ethernet support port. Our API will remain the same as shown in figure 6.3. The specific detailed implementation will change according with the Core/TE device, protocol, and its buffer architecture. The *Device* type is *Ethernet*.

EthernetConnection
+isDeviceOK(Device : DeviceType) : boolean +Read(Device : DeviceType, Message : MessageType*) : boolean +Write(Device : DeviceType, Message : MessageType*) : boolean +EthernetConnection(buffer_size : unsigned short, status : unsigned short*) +~EthernetConnection() +Send() : boolean

Figure 6.3. Core and TE API

6. TIO CSCI and Terminal Exerciser

The Terminal Exerciser can also perform reprogramming and recording of MIDS data from the TIO CSCI. The communication between the TIO CSCI and Terminal Exerciser can be accomplished via the Avionics 1553 Mux. Our API will remain the same as shown in figure 6.4. The specific detailed implementation will change according with the Core/TE device, protocol, and its buffer architecture. The *Device* type is *MIL-STD-1553*.

MIL-STD-1553Connection
+isDeviceOK(Device : DeviceType) : boolean +Read(Device : DeviceType, Message : MessageType*) : boolean +Write(Device : DeviceType, Message : MessageType*) : boolean +MIL-STD-1553Connection(buffer_size : unsigned short, status : unsigned short*) +~MIL-STD-1553Connection() +Send() : boolean

Figure 6.4. TIO and TE API

THIS PAGE INTENTIONALLY LEFT BLANK

VII. CONCLUSIONS & FUTURE WORK

This chapter provides the conclusions of the research conducted and remaining challenges, which require future research in the field of interoperability with legacy software systems.

A. CONCLUSIONS

This thesis proposes an interoperability model, which provides a high-level abstraction for the CSCI interfaces and its interactions to address the re-engineering of old procedural software of the MIDS-LVT to a modern object-oriented architecture. The proposed interoperability model consists of interface, protocol, and temporal specifications. The interface specification is represented by a set of APIs, which act as interfaces for the CSCIs to interact and to cooperate in a distributed heterogeneous environment. The APIs provide a simple programming interface, which shields the software designer from the detailed complex implementation of proprietary device drivers and RTOS facilities. As a result, changes or unsupported proprietary software can be controlled to minimize the impact of the application software.

The protocol specification is a strict constraint mechanism or policy that controls the legal ordering of the sequence of messages involved in the interaction of two CSCIs. The use of the protocol provides a safe and verifiable information exchange between the CSCIs.

For the temporal specification, we are interested in the ability of the system to schedule the functions that provide and consume the data for interaction between two CSCIs. For two CSCIs to be interoperable, their temporal requirements need to be compatible.

These specifications are critical for system interoperability but have not been sufficiently identified in practice. The proposed model is expected to formalize the interoperability requirements for the MIDS-LVT system and to identify and improve the component performance. After being applied in the modernization of the Core CSCI

components, the model can be extended for other CSCI components with correspondent requirement abstractions.

Our experiment showed that the use of API incurred only about two percent of overhead. Based on this result, we conclude that this model provides value added to the effort of re-engineering old procedural software of the MIDS-LVT to a modern object-oriented architecture.

B. FUTURE WORK

Future study should formalize the interoperability model and should consider at least one additional aspect to the model, total system performance response. In a plug-and-play environment, we want to guaranty that the composition of the CSCI components can achieve the robustness, reliable, and maintainable with interchangeable of software and hardware components.

One immediate research area is to experimentally assess the effect of using CORBA as a replacement for shared-memory communication inside the MIDS-LVT.

LIST OF REFERENCES

- [1] Czarnecki K., Eisenecker U., *Generative Programming Methods, Tools, and Applications*, Addison-Wesley, 2000.
- [2] Douglass P., *Doing Hard Time: UML, Objects, Frameworks, and Patterns in Real-Time Software Development*, Addison-Wesley 1999.
- [3] Gomaa H., *Designing Concurrent, Distributed, and real-time Applications with UML*, Addison-Wesley 2000.
- [4] Cho I., McGregor J., Krause L., "A Protocol Based Approach to Specifying Interoperability between Objects," Technology of Object-Oriented language, 1998.
- [5] Rowe D., "Industry Concerns and Problems in Real-Time Object-Oriented Development," Advancement and Research Workshop (COTAR'95), 1995.
- [6] Amza C., Cox A.L, Dwarkadas S., Keleher P., Lu H., Rajamony R., Yu W., Zwaenepoel W., "TreadMarks: Shared Memory Computing on Networks of Workstations," Computer, 1996.
- [7] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns Elements of Reusable Object-Oriented Software*, Addison-Wesley 1995.
- [8] Young P., Berzins V., Ge J., Lugi, "Use of Object Oriented Model for Interoperability in Wrapper-Based Translator for Resolving Representational Differences between Heterogeneous Systems, Engineering Automation for Software Intensive System Integration," June 18-22, 2001.
- [9] Object Management Group, The Common Object Request Broker: Architecture and Specification v.2.4, *OMG Document Number 00.10.01* (2000).
- [10] Polze A., Plakosh D., Wallnau K., "CORBA in Real-Time Setting: A Problem from the Manufacturing Domain," *IEEE Software*, May 1998, pp. 403-411.
- [11] Lankes S., Pfeiffer M., Bemmerl T., "Design and Implementation of a SCI-based Real-Time CORBA," Object-Oriented Real-Time Distributed Computing (ISORC – 2001), February 2001, pp. 23-30.
- [12] Berzins V., Luqi., Shultes B., Guo J., Allen J., Cheng N., Gee K., Nguyen T., Stierna E., "Interoperability Technology Assessment for Joint C4ISR Systems," Naval Postgraduate School Report NPSCS-00-001, September 1999.
- [13] Zaremski A., Wing J., "Specification matching of software components," ACM SIGSOFT Symp. On the Foundations of Software Engineering, October 1995.

- [14] Purtilo J., "The polyolith software bus," ACM Transaction on Programming Languages and Systems, January 1994.
- [15] Dijkstra, E.W. "Co-operating Sequential Process," Programming Languages, F. Genuys, Ed., Academic Press, New York, 1968, pp. 43-112.

APPENDIX A. API LISTING

A. C++ VERSION

1. ShrdMmry.h

```
/**
 * The <code>ShrdMmry</code> class defines the API methods for reading, writing,
 * checking, sending, constructing and destructing shared memory connection for
 * inter-processor communication.
 *
 * @author Theng C. Moua
 * @version 1.0, 10 September 2001
 */

#ifndef ShrdMmry_h
#define ShrdMmry_h 1

// Shared memory manager
#include "ShmMgr.h"

// Message Type
#include "MssgType.h"

// Device Type
#include "DevcType.h"

#define HANDSHAKE          0                /* location of the hand shake word */
#define POINTERS           1                /* pointer starting location */
#define DTBS               8                /* data starting location */

typedef unsigned char      byte;           /* Byte is a char */
typedef unsigned short int word16;        /* 16-bit word is a short int */
typedef unsigned int       word32;        /* 32-bit word is an int */

class Connection
{
public:
    ///# Operation: pure virtual function isDeviceOK
    virtual bool isDeviceOK(DeviceType Device) = 0;

    ///# Operation: pure virtual function Read
    virtual bool Read(DeviceType Device, MessageType* Message) = 0;

    ///# Operation: pure virtual function Write
    virtual bool Write(DeviceType Device, MessageType* Message) = 0;
};

// Class Connection

class SharedMemoryConnection : protected Connection
{
public:
    ///# Constructors (generated)
    SharedMemoryConnection(unsigned short, unsigned short*);

    ///# Destructor (generated)
    ~SharedMemoryConnection();

    ///# Other Operations (specified)
    ///# Operation: isDeviceOK

```

```

    bool isDeviceOK(DeviceType);

    ///## Operation: Read data from shared memory
    bool Read(DeviceType, MessageType*);

    ///## Operation: Write data to shared memory
    bool Write(DeviceType Device, MessageType*);

    ///## Operation: Send data out
    bool Send();

private:
    unsigned short *shm_addr;
    unsigned short shm_size;
    unsigned short pointer_index;
    unsigned short DTB_index;
    unsigned short shm_id;
    bool shm_status;
    unsigned short current_shm_size;

    ///## Operation: isFinishedRead
    bool isFinishedRead();

    ///## Operation: isFinishedWrite
    bool isFinishedWrite();

    ///## Operation: setFinishedWrite
    bool setFinishedWrite();

    ///## Operation: setFinishedRead
    bool setFinishedRead();

    ///## Operation: getPointer
    bool getPointer(unsigned short*);

    ///## Operation: setInterrupt
    bool setInterrupt();

    ///## Operation: hasNoMoreMessage
    bool hasNoMoreMessage();

    ///## Operation: hasNoRoomForMessage
    bool hasNoRoomForMessage(MessageType*);
};

// Class SharedMemoryConnection

#endif

```

2. ShrdMmry.cpp

```

/**
 * The <code>ShrdMmty</code> class implements the API methods for reading, writing,
 * checking, sending, constructing and destructing shared memory connection for
 * inter-processor communication.
 *
 * @author Theng C. Moua
 * @version 1.0, 10 September 2001
 */

// SharedMemoryConnection

#include "ShrdMmry.h"
#include <iostream.h>

// Class SharedMemoryConnection

```



```

/** Constructor for SharedMemoryConnection
 * @param buffer_size the size of memory region declared in unsigned short (16 bit word)
 * @param status is the status pass by pointer to allocate of a shared memory region. The
 * return value indicate success (1) or failure (0).
 */

SharedMemoryConnection::SharedMemoryConnection(unsigned short buffer_size, unsigned
short* status)
{
    cout << "Sharedmem const\n";
    unsigned short allocate_status;

    shm_addr = ShmManager::allocate( buffer_size, &allocate_status);

    if (!allocate_status)
    {
        *status = 0;
        cout << " Error condition, can't allocate memory \n";
    }
    else
    {
        shm_size = buffer_size;
        pointer_index = 0;
        DTB_index = 0;
        *status = 1;
        current_shm_size = 0;
        shm_id = ShmManager::getTotalShmObject(); //get the current number
    }
}

/** Destructor for SharedMemoryConnection
 * Parameters are not required
 */

SharedMemoryConnection::~SharedMemoryConnection()
{
    delete [] shm_addr;
}

//### Other Operations (implementation)

/** Check the SharedMemoryConnection
 * @param Device is the device type (Sharedmem) required for this method
 * @return <tt>true</tt> if internal states are successfully check.
 */

bool SharedMemoryConnection::isDeviceOK(DeviceType Device)
{
    unsigned short dpmidx;
    bool status = true;

    if (Device != Sharedmem)
        return false;

    for (dpmidx = 0; dpmidx < shm_size; dpmidx++)
        shm_addr[dpmidx] = 0x0000;

    dpmidx = 0x0000;

    do
    {
        if (shm_addr[dpmidx] != 0x0000)
        {
            status = false;
        }
    }
    else

```

```

        {
            dpmidx++;
        }

    }while ((dpmidx < shm_size) && (status == true));

    return status;
}

/** Read message from SharedMemoryConnection
 * @param Device is the device type (Sharedmem) required for this method
 * @param Message is a message type required for read and write to the memory region
 * @return <tt>true</tt> if internal states are sucessfully read.
 */

bool SharedMemoryConnection::Read( DeviceType Device,  MessageType* Message)
{
    unsigned short *dtb;
    unsigned short offset;

    if (Device != Sharedmem)
        return false;

    if (hasMoreMessage())
    {
        getPointer(&offset);
        dtb = shm_addr + offset;
        Message->readDTB(dtb);
        shm_addr[pointer_index] = 0x0000;

        return true;
    }
    else
    {
        if (!(setFinishedRead()))
            cout << "Error, can't reset the lock\n";

        return false;
    }
}

/** Check for no more room in the SharedMemoryConnection
 * @return <tt>true</tt> if internal states are sucessfully check no more room.
 */

bool SharedMemoryConnection::hasRoomForMessage(MessageType* Message)
{
    if ( (current_shm_size + Message->getSize()) > shm_size )
        return false;
    else
        return true;
}

/** Check for no more message in the SharedMemoryConnection
 * @return <tt>true</tt> if internal states are sucessfully check no more message.
 */

bool SharedMemoryConnection::hasMoreMessage()
{
    if ((shm_addr[POINTERS + pointer_index] == 0) || (pointer_index == 7))
        return false;
    else
        return true;
}

/** Write message to SharedMemoryConnection

```

```

* @param Device is the device type (Sharedmem) required for this method
* @param Message is a message type required for read and write to the memory region
* @return <tt>true</tt> if internal states are sucessfully write.
*/

bool SharedMemoryConnection::Write( DeviceType Device,  MessageType* Message)
{
    unsigned short *pointer, *dtb;

    if (Device != Sharedmem)
        return false;

    if ( !(hasRoomForMessage(Message)) )
    {
        cout<< "Error condition, no more room in the shared memory \n";
        return false;
    }

    if (isFinishedRead())
    {
        /* pointer to memory pointer area */
        pointer = shm_addr + POINTERS + pointer_index;

        *pointer = DTBS + DTB_index;          /* value of next data location */

        dtb = shm_addr + *pointer;
        Message->writeDTB(dtb);
        DTB_index = DTB_index + Message->getSize();

        pointer_index++;

        // set the current size of the shared memory
        current_shm_size += Message->getSize();

        return true;
    }
    else
    {
        cout<< "Error condition, not Finished Read yet \n";
        return false;
    }
}

/** Check if receiver finished reading from SharedMemoryConnection
* @return <tt>true</tt> if internal states are sucessfully check for finished reading.
*/

bool SharedMemoryConnection::isFinishedRead()
{
    if (shm_addr[HANDSHAKE] == 0x0000)
        return true;
    else
        return false;
}

/** Check if receiver finished writing from SharedMemoryConnection
* @return <tt>true</tt> if internal states are sucessfully check for finished writing.
*/

bool SharedMemoryConnection::isFinishedWrite()
{
    if (shm_addr[HANDSHAKE] == 0x0001)
        return true;
    else
        return false;
}

```

```

/** Set finished writing to the SharedMemoryConnection
 * @return <tt>true</tt> if internal states are sucessfully set finished writing.
 */

bool SharedMemoryConnection::setFinishedWrite()
{
    DTB_index = 0;
    pointer_index = 0;

    shm_addr[HANDSHAKE] = 0x0001;

    if (shm_addr[HANDSHAKE] == 0x0001)
        return true;
    else
        return false;
}

/** Set finished reading to the SharedMemoryConnection
 * @return <tt>true</tt> if internal states are sucessfully set finished reading.
 */

bool SharedMemoryConnection::setFinishedRead()
{
    pointer_index = 0;

    shm_addr[HANDSHAKE] = 0x0000;

    if (shm_addr[HANDSHAKE] == 0x0000)
        return true;
    else
        return false;
}

/** Get pointer from SharedMemoryConnection
 * @param offset is pass by pointer, the return value is the address of the next message
 * @return <tt>true</tt> if internal states are sucessfully write.
 */

bool SharedMemoryConnection::getPointer(    unsigned short *offset)
{
    *offset = shm_addr[ POINTERS + pointer_index];
    pointer_index++;

    return true;
}

/** Send messages from SharedMemoryConnection
 * @return <tt>true</tt> if internal states are sucessfully send.
 */

bool SharedMemoryConnection::Send()
{
    if (setFinishedWrite() == false)
    {
        cout << "Error, can't free the lock\n";
        return false;
    }

    if (setInterrupt() == false)
    {
        cout << "Error, can't set the interrupt\n";
        return false;
    }

    current_shm_size = 0;

    return true;
}

```

```

}

/** Set interrupt to SharedMemoryConnection
 * @return <tt>true</tt> if internal states are sucessfully set interrupt.
 */

bool SharedMemoryConnection::setInterrupt()
{
    // implementation of set the software interrupt (DTI)
    return true;
}

// main program uses to test the APIs

int main()
{
    unsigned short status, shm_max_size;
    unsigned short size_dtb1 = 34;
    unsigned short size_dtb2 = 34;
    unsigned short size_dtb3 = 34;
    unsigned short size_dtb4 = 34;
    unsigned short size_dtb5 = 34;
    unsigned short size_dtb6 = 131;
    unsigned short size_dtb7 = 204;
    shm_max_size = 1024;

    ShmManager* myshmgr;
    myshmgr = new ShmManager();

    SharedMemoryConnection* myshrmem;

    myshrmem = new SharedMemoryConnection(shm_max_size, &status);

    if (!status)
    {
        cout << status << " Error, Device fail the allocation and init check \n";
        delete myshrmem;
    }

    MessageType* dtb1_out;
    dtb1_out = new MessageType( size_dtb1, DTB1);

    MessageType* dtb2_out;
    dtb2_out = new MessageType( size_dtb2, DTB2);

    MessageType* dtb3_out;
    dtb3_out = new MessageType( size_dtb3, DTB3);

    MessageType* dtb4_out;
    dtb4_out = new MessageType( size_dtb4, DTB4);

    MessageType* dtb5_out;
    dtb5_out = new MessageType( size_dtb5, DTB5);

    MessageType* dtb6_out;
    dtb6_out = new MessageType( size_dtb6, DTB6);

    MessageType* dtb7_out;
    dtb7_out = new MessageType( size_dtb7, DTB7);

    // input dtbs

    MessageType* dtb_in;
    dtb_in = new MessageType( shm_max_size);

    MessageType* dtb1_in;
    dtb1_in = new MessageType( size_dtb1, DTB1);

```

```

MessageType* dtb2_in;
dtb2_in = new MessageType( size_dtb2, DTB2);

MessageType* dtb3_in;
dtb3_in = new MessageType( size_dtb3, DTB3);

MessageType* dtb4_in;
dtb4_in = new MessageType( size_dtb4, DTB4);

MessageType* dtb5_in;
dtb5_in = new MessageType( size_dtb5, DTB5);

MessageType* dtb6_in;
dtb6_in = new MessageType( size_dtb6, DTB6);

MessageType* dtb7_in;
dtb7_in = new MessageType( size_dtb7, DTB7);

// write data out to shared memory

if (!(myshrmem->Write( Sharedmem, dtb1_out)))
    cout << status << " Error, write fail \n";

if (!(myshrmem->Write( Sharedmem, dtb2_out)))
    cout << status << " Error, write fail \n";

if (!(myshrmem->Write( Sharedmem, dtb3_out)))
    cout << status << " Error, write fail \n";

if (!(myshrmem->Write( Sharedmem, dtb4_out)))
    cout << status << " Error, write fail \n";

if (!(myshrmem->Write( Sharedmem, dtb5_out)))
    cout << status << " Error, write fail \n";

if (!(myshrmem->Write( Sharedmem, dtb6_out)))
    cout << status << " Error, write fail \n";

if (!(myshrmem->Write( Sharedmem, dtb7_out)))
    cout << status << " Error, write fail \n";

if (!(myshrmem->Send()))
    cout << status << " Error, send fail \n";

// read data from shared memory

while (myshrmem->Read( Sharedmem, dtb_in) )
{
    switch (dtb_in->getRecvMsgID())
    {
        case DTB1:
            dtb_in->copyBuffer(dtb1_in->getBufferAddress(), size_dtb1);
            break;
        case DTB2:
            dtb_in->copyBuffer(dtb2_in->getBufferAddress(), size_dtb2);
            break;
        case DTB3:
            dtb_in->copyBuffer(dtb3_in->getBufferAddress(), size_dtb3);
            break;
        case DTB4:
            dtb_in->copyBuffer(dtb4_in->getBufferAddress(), size_dtb4);
            break;
        case DTB5:
            dtb_in->copyBuffer(dtb5_in->getBufferAddress(), size_dtb5);
            break;
        case DTB6:
            dtb_in->copyBuffer(dtb6_in->getBufferAddress(), size_dtb6);
            break;
        case DTB7:

```

```

        dtb_in->copyBuffer(dtb7_in->getBufferAddress(), size_dtb7);
        break;
    default:
        break;
    }
}

// test second shared memory

SharedMemoryConnection* myshrmem1;
myshrmem1 = new SharedMemoryConnection(shm_max_size, &status);

if ( status)
    status = myshrmem1->isDeviceOK( Sharedmem );
else
    delete myshrmem1;

// test third shared memory

SharedMemoryConnection* myshrmem2;

myshrmem2 = new SharedMemoryConnection(shm_max_size, &status);

if ( status)
    status = myshrmem2->isDeviceOK( Sharedmem );
else
    delete myshrmem2;

// how many shared memory object created
cout << "Total is " << ShmManager::getTotalShmObject() << "\n";

return 0;
}

```

3. MssgType.h

```

/**
 * The <code>MssgType</code> class defines the API methods for reading, writing,
 * getting info, constructing and destructing message buffer to and from the shared
 * memory region.
 *
 * @author Theng C. Moua
 * @version 1.0, 10 September 2001
 */

#ifndef MssgType_h
#define MssgType_h 1

#include <iostream.h>
#include <stdlib.h>

typedef enum {DTB0,DTB1, DTB2, DTB3, DTB4, DTB5, DTB6, DTB7} MsgIDType;

class MessageType
{
public:
    /// Constructors (generated)
    MessageType(unsigned short, MsgIDType);

    /// Constructors ( temp read buffer)
    MessageType(unsigned short);

    /// Destructor (generated)

```

```

~MessageType();

///  

Operation: ReadDTB  

bool readDTB (unsigned short*);

///  

Operation: WritDTB  

bool writDTB (unsigned short*);

///  

Operation: CopyBuffer  

bool copyBuffer (unsigned short*, unsigned short);

///  

Operation: getSize  

unsigned short getSize();

///  

Operation: getRecvMsgID  

unsigned short getRecvMsgID();

///  

Operation: getBufferAddress  

unsigned short* getBufferAddress();

private:
    // Data Members for Class Attributes
    unsigned short Size;
    unsigned short* msg;
    unsigned short Status;
    MsgIDType ID;
};

// Class MessageType

#endif

```

4. MssgType.cpp

```

// MessageType
#include "MssgType.h"

// Class MessageType

///  

Constructors (generated)

/** Constructor for MessageType
 * @param m_size is the size of the message buffer (DTB) declare as unsigned 16 bit word
 * @param m_id is the type of DTB declare as enum
 */

MessageType::MessageType(unsigned short m_size, MsgIDType m_id)
{
    int index;

    Size = m_size;
    msg = new unsigned short [Size];

    msg [0] = m_id;
    msg [1] = Size;

    for (index = 2; index < Size; index++)
    {
        msg [index] = (unsigned short) rand();        // simulate data
    }
}

///  

Constructors ( temp read buffer)

/** Constructor for MessageType
 * @param m_size is the size of the temp message buffer (DTB)
 */

```



```

MessageType::MessageType(unsigned short m_size)
{
    int index;

    Size = m_size;
    msg = new unsigned short [Size];

    for (index = 0; index < Size; index++)
    {
        msg [index] = 0;
    }
}

//## Destructor (generated)

/** Destructor for MessageType
 * Parameter are not required
 */

MessageType::~MessageType()
{
    delete msg;
}

/** Read the MessageType
 * @param source is the address of the DTB in the shared memory region
 * @return <tt>true</tt> if internal states are successfully read.
 */

bool MessageType::readDTB (unsigned short* source)
{
    unsigned short size;
    unsigned short *dest;

    dest = msg;
    size = source[1];    //size

    while ( size > 0)
    {
        *dest++ = *source++;
        size--;
    }

    return true;
}

/** Write the MessageType
 * @param dest is the address of the DTB in the shared memory region
 * @return <tt>true</tt> if internal states are successfully write.
 */

bool MessageType::writeDTB (unsigned short* dest)
{
    unsigned short size;
    unsigned short *source;

    source = msg;
    size = Size;

    int i = 0;

    while ( size > 0)
    {
        i++;
        *dest++ = *source++;
        size--;
    }
}

```

```

        return true;
    }

    /** Copy from the MessageType
    * @param sest is the address of the DTB in the shared memory region
    * @param size is the size of the buufer to be copy
    * @return <tt>true</tt> if internal states are successfully copy.
    */

bool MessageType::copyBuffer (unsigned short* dest,unsigned short size )
{
    unsigned short *source;
    int i = 0;

    source = msg;

    while ( size > 0)
    {
        i++;
        *dest++ = *source++;
        size--;
    }

    return true;
}

/** Get size from the MessageType
* @return the size of the message buffer
*/

unsigned short MessageType::getSize()
{
    return Size;
}

/** Get message ID from the MessageType
* @return the ID of the message buffer
*/

unsigned short MessageType::getRecvMsgID()
{
    return msg[0];
}

/** Get address from the MessageType
* @return the address of the message buffer
*/

unsigned short* MessageType::getBufferAddress()
{
    return msg;
}

```

5. ShmMgr.h

```

/**
* The <code>ShrMgr</code> class implements the API methods for allocating, getting info,
* constructing and destructing shared memory region for
* inter-processor communication.
*
* @author Theng C. Moua
* @version 1.0, 10 September 2001
*/

#ifndef ShmManager_h
#define ShmManager_h 1

```

```

const int MAX_MEM_SIZE = 16000;

class ShmManager
{
public:
    /// Constructors (generated for simulation purpose)

    /** Constructor for ShmManager
     * Parameters are not required
     */

    ShmManager()
    {
        base_addr = new unsigned short [MAX_MEM_SIZE];

        current_addr = base_addr;    // this is the VME-base address
        current_size = 0;
        count = 0;
    };

    /// Destructor (generated)

    /** Destructor for ShmManager
     * Parameters are not required
     */

    ~ShmManager()
    {
        delete [] base_addr;
    };

    /** Allocate shared memory region from ShmManager
     * @param m_shm_size is the size of the shared memory region requesting
     * @param status is the status pass by pointer to allocate of a shared memory
     * region. The return value indicate success (1) or failure (0).
     */

    static unsigned short* allocate(unsigned short m_shm_size, unsigned short* status)
    {
        if ( (current_size + m_shm_size) > MAX_MEM_SIZE )
            *status = 0;                                //fail
        else
        {
            current_size += m_shm_size;
            current_addr += m_shm_size;
            *status = 1;                                //success
            count++;
        }

        return current_addr;
    };

    /** Get number of region from ShmManager
     * @return <tt>true</tt> if internal states are successfully get the info.
     */

    static unsigned short ShmManager::getTotalShmObject()
    {
        return count;
    }

private:    /// implementation
    // Data Members for Class Attributes

    unsigned short          *base_addr;
    static unsigned short    *current_addr;
    static unsigned int      current_size;
    static unsigned short    count;

```

```
};

// Class ShmManager

    unsigned short ShmManager::count = 0;           // definition of count
    unsigned short* ShmManager::current_addr; // definition of current_address
    unsigned int ShmManager::current_size = 0; // definition of current_size

#endif
```

6. DevcType.h

```
/**
 * The <code>DevcType</code> class defines the type of devices for the
 * inter-processor communication.
 *
 * @author Theng C. Moua
 * @version 1.0, 10 September 2001
 */

typedef enum { Sharedmem, DMA, Ethernet, Serial, Mil_1553 } DeviceType;
```

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, VA 22060-6218
2. Dudley Knox Library
Naval Postgraduate School
Monterey, CA 93943-5101
3. Dr Luqi
Naval Postgraduate School
Monterey, California 93943-5118
luqi@cs.nps.navy.mil
4. Dr Valdis Berzins
Naval Postgraduate School
Monterey, California 93943-5118
berzins@cs.nps.navy.mil
5. Dr Ge Jun
Naval Postgraduate School
Monterey, California 93943-5118
gejun@cs.nps.navy.mil
6. Mr. Theng Moua
San Diego, California 92111
moua@spawar.navy.mil
7. SPAWAR, PMW/PMA-150
San Diego, California 92110-3215
eastl@spawar.navy.mil